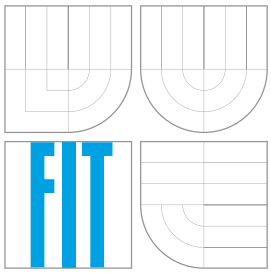
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

MODERN METHODS OF REALISTIC LIGHTING IN REAL TIME

MODERNÍ TECHNIKY REALISTICKÉHO OSVĚTLENÍ V REÁLNÉM ČASE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. ISTVÁN SZENTANDRÁSI

SUPERVISOR
VEDOUCÍ PRÁCE
doc. Ing. ADAM HEROUT, Ph.D.

BRNO 2011

Abstract

Physically plausible illumination in real-time is often achieved using approximations. Recent methods approximate global illumination in the screen space by exploiting the capabilities of modern graphics cards. Two of these techniques, screen-space ambient occlusion and screen-space directional occlusion, are described in this work. Screen-space directional occlusion is a generalized version of screen-space ambient occlusion. It supports one indirect bounce of diffuse light and depends on the direction of incoming light. The main goal of this project is to further experiment with these methods and improve them. For a uniform distribution of the sampling points, the Halton sequence is used. In order to reduce the noise, geometry-aware bilateral filtering is applied. Methods are further sped up by computing them in a lower resolution, and they are restored to full resolution using joint bilateral upsampling in order to create the final image. Apart from global illumination techniques some shadow mapping techniques and high dynamic range rendering is discussed, too.

Abstrakt

Fyzikálně přijatelné osvětlení v reálném čase je často dosaženo použitím aproximací. Současné metody často aproximují globální osvětlení v prostoru obrazu s využitím schopností moderních grafických karet. Dva techniky z této kategorie, screen-space ambient occlusion a screen-space directional occlusion jsou popsány detailněji v této práci. Screen-space directional occlusion je zobecněná verze screen-space ambient occlusion s podporou jednoho difúzního odrazu a závislostí na směrové informaci světla. Hlavním cílem projektu bylo experimentování s těmito metodami. Pro uniformní distribuci náhodných vzorek pro obě metody byla použita Halton sekvence. Pro potlačení šumu je použita bilaterální filtrace, která bere do úvahy geometrické vlastnosti scény. Metody jsou dál zrychleny použitím nižších rozlišení pro výpočet. Rekonstrukce výsledků do původní velikosti pro vytvoření konečného obrazu je realizována pomoci joint bilateral upsamplingu. Kromě metod globálního osvětlení byly v práci použity aj metody pro mapování stínů a HDR osvětlení.

Keywords

Global illumination, screen-space ambient occlusion, screen-space directional occlusion, Halton sequence, bilateral filtering, subsampling, joint bilateral upsampling, shadow mapping, high dynamic range rendering

Klíčová slova

Realistické osvětlení, screen-space ambient occlusion,screen-space directional occlusion, Halton sekvence, bilaterální filtrování, podvzorkování, joint bilateral upsampling, mapování stínů, hdr osvětlení

Citace

István Szentandrási: Modern Methods of Realistic Lighting in Real Time, diplomová práce, Brno, FIT VUT v Brně, 2011

Modern Methods of Realistic Lighting in Real Time

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Ing. Adam Herout, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
István Szentandrási
25. května 2011

Poděkování

Chtěl bych poděkovat mému vedoucímu doc. Ing. Adam Herout, Ph.D. za jeho připomínky a cenné rady, poskytnutou odborní pomoc a trpělivost.

© István Szentandrási, 2011.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

Chapter 1

Introduction

Computing global illumination in real time has been and still is a major challenge in computer graphics. Due to the complexity of light transport and some material properties, real-time frame rates can only be achieved at the cost of trade-offs and rough approximations. There already exist a few methods that simulate some lighting properties realistically. Direct diffuse lighting, hard shadows and high dynamic range rendering are good examples. The simulation of these lighting effects usually form the core of illumination computations for all 3D rendering engines with the goal of creating realistic images.

Using just the aforementioned techniques still causes images to look highly artificial. Adding global illumination could improve the quality significantly. Perceptually among the most important optical phenomena belong soft shadows and indirect lighting. There have been many attempts to simulate either of these in real time. A handful of these attempts are based on ambient occlusion (AO) [25], which is very popular in the film industry as well as in games. The main advantage lies in their speed and simple implementation.

As in every approximation, ambient occlusion has some limitations, too. The basic method [25] displays darkening of cavities; however it does not take into account the direction and intensity of light coming from light sources or environmental maps. A better method has been introduced by Ritschel et al. [19] called screen-space directional occlusion (SSDO). It provides more realistic illumination: it accounts for the direction of the incoming light and supports a single indirect bounce of light.

The aim of this project is to give a brief overview of the already existing techniques and to experiment with them. The main focus is on the global illumination methods: screen-space ambient and directional occlusion. Since screen-space directional occlusion is a fairly new technique, and screen-space ambient occlusion is already a well known and used method, screen-space directional occlusion stands in center of the experimentations, possibly leading to improvements. Screen-space ambient occlusion is used mainly as a reference for image quality and speed.

The work is structured as follows. In the first two chapters (Chapter 2 and Chapter 3) I present a few methods used for rendering visually convincing images. Chapter 4 contains detailed description of the experiments and improvements for screen-space directional occlusion. Chapter 5 describes the design of the implementation for the testing application. Finally the achieved results and comparison between methods is discussed in Chapter 6.

Chapter 2

Realistic Lighting in Real Time

The effort to render images indistinguishable from pictures taken of real scenes is one of the main forces that move computer graphics forwards. The most obvious approach is to simulate the physical model of light transport and material properties. Historically the two main techniques are ray tracing and radiosity. However, both methods require massive computations.

Ray tracing based methods model the path and energy carried by photons through a scene. It supports among other things advanced material properties, hard shadows, reflections. This is a very robust approach, since it is practically an exact model of the physical lighting properties. Ray tracing is usually computed offline. Major slowing factors for these methods are the number of object-ray intersections and visibility tests. When computing more advanced lighting effects, for example fog, subsurface scattering, soft shadows or using area light sources causes the number of rays to grow exponentially, hence slowing down the method even more. There are many possibilities to accelerate methods based on ray tracing: final gathering, irradiance caching, sparse sampling, advanced space division structures, etc. Recently, with the growth in the speed of the processing units and the introduction of GPGPU solutions, it is possible to compute illumination to some extent using ray tracing in real time. The quality of the images rendered in real time unfortunately still lags behind the images rendered using hardware accelerated methods using OpenGL or DirectX.

Unlike ray tracing, radiosity only accounts for diffuse light. Radiosity is based on solving a system of equations representing the transfer of radiosity between patches in the scene. It supports area lights, soft shadows, indirect lighting. However, it is currently impossible to compute a complete solution for the whole scene in real time.

Nowadays, the dominant way to render visually convincing images in real time is using OpenGL, DirectX or other similar API leveraging hardware acceleration. These methods do not try to model physical lighting precisely. Their aim is just to render images, that "look good". The observations and knowledge from ray tracing and radiosity are sometimes used to make images visually more convincing. Hard and partially soft shadows from a directional or point light is currently mostly done using shadow mapping (Section 2.1). Some effects from radiosity and ray tracing, like indirect lighting, soft shadows, are approximated using global illumination techniques (Chapter 3).

2.1 Shadow Mapping

Shadow mapping [24] is an algorithm to create shadows caused by point and directional lights in real-time. There are currently many variations to shadow mapping, but the basic idea for all is the same.

In a nutshell, shadows are computed the following way. The algorithm can be split into two main parts. First the scene is rendered for each light from the light's point of view. The information from this pass is usually saved in a depth map representing the distance of the scene surface from the given light source for each pixel. In the second part the scene is rendered from the camera's point of view. In this part the distances between the surface points represented by pixels and the light sources are computed. These distances are then compared to the values stored in the depth map for each pixel to decide, whether the pixel is in shadow or lit by the light source (Figure 2.1[1]).

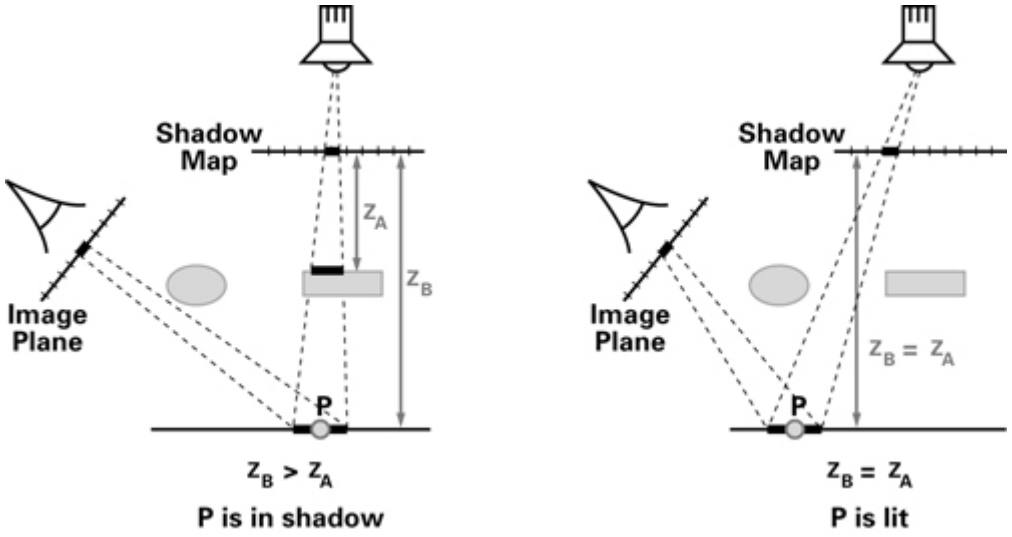


Figure 2.1: Shadow mapping principle. The distances from the light source are saved in the shadow map. For each pixel in the image space the distance from the light is compared to the value in the shadow map. If the distance is bigger (left picture) the pixel is in shadow, otherwise (right image) the pixel is lit.

Since the depth comparison and rendering from different views can all be achieved using hardware acceleration, this technique is really fast. The depth comparison is done on the pixel level, therefore self-shadowing for objects is not a problem. Another advantage of shadow mapping over other shadowing techniques, that it can be used for transparent objects, where the transparency is defined in the texture.

Shadow mapping has some disadvantages, too. These disadvantages are usually caused by the limited resolution of the depth maps (commonly stored as a texture), where the distances from the lights are stored for each pixel.

Bias

The source of the first problem comes from the fact, that the depth values are stored only for the pixel center. The projected position in the depth map of the surface points might

[1] http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html

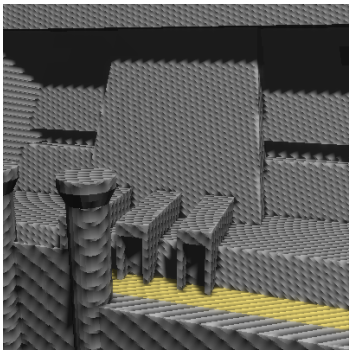


Figure 2.2: Shadow mapping artifacts without using bias and jagged edges of the shadows.

be between these pixel centers in the second part of the algorithm. Based on the angle between the surface's normal and light direction, the depth stored in the shadow map might be larger or smaller than the computed distance for the surface point in the pixel center from the camera's viewpoint. This leads to self-occlusion artifacts (Figure 2.2).

Let us consider the texels in the shadow map as small patches on surface geometry oriented towards the light source. On surfaces, where the normal is not parallel with the light direction, one half of the patch will be below and the other half above the real surface. The parts above the surface cause the aforementioned self-occlusion. This problem can be solved by introducing a bias to the shadow map values, so that the patches are completely below surface. The minimal size for this bias is $b = \frac{p}{2} \tan \alpha$, where p is the size of the patches and α is the angle between the surface normal and light direction.

Percentage Closer Filtering

As the shadow maps are stored in textures and have limited resolution, aliasing appears on the edges of the shadows (Figure 2.2). This problem can be solved with percentage-closer filtering. [16]

Unlike normal textures simple shadow map textures cannot be prefiltered to remove aliasing. Instead of smoothing the textures, multiple shadow map comparisons are made. The results of the comparisons are then averaged together (Figure 2.3). The method is called "percentage-closer", because it calculates the percentage of the surface which is not in shadow. Using this method an area of penumbra appears. This imitation of soft shadow may further improve perception.

Variance Shadow Mapping

Variance shadow mapping similarly to percentage closer filtering computes a probability of the pixel being in shadow. It is a fairly new technique. It was introduced in 2006 by Donnelly et al. [6].

Variance shadow mapping algorithm instead of saving a single depth value in the first pass from the lights viewpoint, stores also the square of the depth. Another difference is that while with normal shadow mapping prefiltering of the shadow map is undesirable; with variance shadow mapping it is beneficial. The shadow map can be filtered and smoothed with arbitrary size kernel without a major impact on the performance (unlike percentage closer filtering), for example separable gaussian blur.

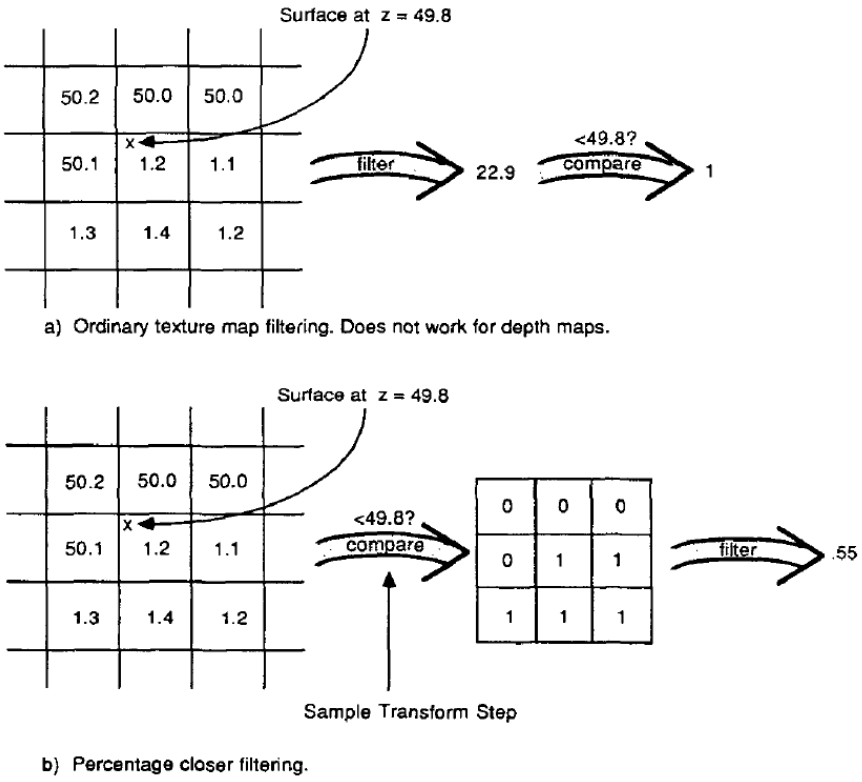


Figure 2.3: Percentage closer filtering principle. [16]

Based on the two filtered values stored in the shadow map, the mean μ and variance σ^2 is computed for the area effected by filtering. These values are then used to compute an upper bound of the probability of the pixel being occluded using Chebychev's inequality[2] [6].

Other Methods

There are more shadow mapping methods that try either to improve the quality of the rendered shadows or speed them up. For example convolution shadow maps, exponential shadow maps, percentage closer soft shadows, etc. [1]

Larger scenes for shadow mapping in general still represent a huge issue. The goal is to get adequately good results for every part of the scene. Perspective shadow maps [21] transform the viewing projection from the light to focus on the visible part of the scene. Cascaded shadow maps [4] use multiple shadow maps for different parts of the scene to achieve good quality shadows everywhere. Since neither of these methods were used in this project, for a description of these techniques see the appropriate sources.

[2]Chebychev's inequality, one tailed version. Let x be a random variable drawn from a distribution with mean μ and variance σ^2. Then for $t > \mu$: $P(x \geq t) \leq p_max(t) \equiv \frac{\sigma^2}{\sigma^2+(t-\mu)^2}$

2.2 High Dynamic Range Rendering

In order to make rendered images more realistic, a set of techniques gained more attention recently, high dynamic range rendering. Ordinary displays and projectors have a low contrast ratio; much lower than a natural scene exposed in sunlight. High dynamic range rendering allows preserving some details that might be lost due to this limitation. The operation of converting a high dynamic range image to a low dynamic range image is called tone mapping.

Tone Mapping

One of the simplest and most well known global tone mapping operator was introduced by Reinhard et al. [18]. The algorithm first does a global scaling based on the log-average luminance given by:

$$\overline{L_w} = exp(\frac{1}{N}\sum_{x,y}\ln(\delta + L_w(x,y)), \tag{2.1}$$

where N is the number of pixels, L_w is the world luminance of each pixel and δ is a small constant used to avoid numerical underflow. The scaling is then done using a "key value" α, indicating whether the image is subjectively light or dark:

$$L(x,y) = \frac{\alpha}{\overline{L_w}}L_w(x,y). \tag{2.2}$$

A normal "key value" usually is around 0.18. It may vary from 0.045 up to 0.9. A simple tone mapping operator is then given by:

$$L_d(x,y) = \frac{L(x,y)}{1+L(x,y)}. \tag{2.3}$$

This way all luminances are scaled into the $(0,1)$ interval. Sometimes this is not desirable, so an alternative equation can be used, that allows too bright luminances to be clamped to white:

$$L_d(x,y) = L(x,y)\frac{1+\frac{L(x,y)}{L_{white}^2}}{1+L(x,y)}, \tag{2.4}$$

where L_{white} is the smallest luminance that will be mapped to white.

The author of the method notes that while this method works sufficiently for many high dynamic range images, details can be lost for very high dynamic range; mostly for the brighter regions. In order to solve this issue, he used a local tone reproduction algorithm that applies "dodging-and-burning" (locally brightening darker regions and darkening brighter regions) [18].

There are several other tone mapping operators: logarithmic mapping, histogram adjustment, methods using bilateral filtering, local eye adaptation, etc. Since I used bilateral filtering for smoothing in this project (Section 3.5), bilateral filtering based method could have been used. However these methods require quite large filtering kernels. Durand et al. [7] states that a sufficient size for a filtering kernel is 2% of the image size. However, for a full HD image 1920x1080 2% gives a minimum size of 39 pixels sized kernel. Filtering with these kernels could represent a huge performance hit, even on recent graphics cards, thus unattractive for real-time applications.

Bloom Effect

Bloom or glow is an effect used usually with high dynamic range rendering to reproduce an imaging artifact of real cameras. In pictures taken by real-word cameras light bleeding occurs around very bright regions. The artifact is caused by diffraction. Since the lenses are imperfect, the light coming from a given direction does not focus into a perfect point, but to a finite-sized disk with much less noticeable rings around it (Airy disk).

For high dynamic range rendering this effect is reproduced by applying a Gaussian blur before converting the image to low dynamic range. Sometimes, to avoid too much blurring even for darker regions, a threshold is used to apply the blurring only to the bright pixels. [11]

Chapter 3

Global Illumination

The term global illumination does not have an established meaning. A formal definition would be a method for solving the rendering equation. Thanks to the rendering equation, "all light transport mechanisms are described using a recursive integral equation, whose kernel contains the various material properties and the visibility function". [9]

The term in a more general sense usually refers to a group of techniques used in 3D rendering to make images more realistic: soft shadows, indirect lighting, caustics, reflections, etc. Another way of putting this is that global illumination techniques add something extra to an image rendered simply with direct illumination from light sources. In real-time rendering these additional effects usually just mimic the behavior of real-world lighting. A complete physical simulation of light transport would require too much time and resources.

3.1 Previous Work

Since the introduction of ambient occlusion [3][25], it has been widely adopted both in the game and the film industry. Ambient occlusion computes the visibility of the hemisphere at each point of the scene. In order to approximate visibility, the algorithm samples the hemisphere in every direction to compute a darkening factor:

$$AO = \frac{1}{\pi} \int_{\Omega} V(\omega)(\mathbf{N} \cdot \omega)\, \Delta\omega, \tag{3.1}$$

where ω is the direction, V is the visibility test, $\mathbf{N}$ is the normal and Ω is the hemisphere. The method is often calculated by casting rays in every direction over the hemisphere using Monte Carlo sampling. The calculated factor is used to modulate ambient lighting, just as the name suggests. [9]

Casting rays in every direction still requires too much computing power, so a few alternative methods were introduced. These methods compute ambient occlusion less accurately in order to achieve higher frame rates. Instead of computing occlusion over surfaces in 3D, these methods frequently approximate ambient occlusion in the screen space [20][14][2][10]. Screen-space ambient occlusion (SSAO) is very popular due to its simplicity and speed. It does not require any additional data and can be applied as a post-process to the scene. Screen-space ambient occlusion is going to be discussed in more detail in Section 3.2.

Ambient occlusion is just a rough approximation of general light transport. It does not take into account any directional information of the incoming radiance or other more expensive illumination effects (inter-reflections, caustics, subsurface scattering). A different

family of techniques, the precomputed radiance transfer (PRT) [9], does support the aforementioned features. On the other hand, PRT algorithms typically assume static scenes, distant lights or environmental maps.

Screen-space directional occlusion (SSDO) [19] tries to combine the speed and simplicity of screen-space ambient occlusion methods with directional information of lighting and near field indirect color bleeding. More on screen-space directional occlusion is in Section 3.3.

In order to avoid some limitations of screen-space ambient occlusion, a hybrid method was introduced by Reinbothe et al. [17]. This method works in 3D space by voxelization of the scene, calculating occlusion based on this information and finally using bilateral filtering in the screen space to smooth the shadows.

A completely different approach was taken by Kaplanyan et al. [12]. They approximate indirect illumination in fully dynamic scenes using cascaded light propagation volumes. This method supports single bounce illumination with occlusion, but it can be extended to support multiple bounces and to handle participating media.

These techniques show that in order to generate visually convincing images, no physically precise computations are needed. Simple approximations using soft shadows, ambient occlusion, and optionally, a single bounce of indirect light can give out acceptable results even in real time.

3.2 Screen-space Ambient Occlusion

Screen-space ambient occlusion is a coarse approximation of ambient occlusion that works in the screen space in order to achieve real-time frame rates. The idea behind screen-space ambient occlusion is to reuse the z-buffer data, which was already computed during the rendering of the scene. The algorithm samples an area surrounding the pixel. The depths for the matching sampling points are accessible from the z-buffer. The method then computes an average visibility property, a darkening factor, based on the depth comparisons between the samples and the pixel's depth. Analogously to ambient occlusion the darkening factor is used to tone down the light intensity in cavities and corners. In most cases, the darkening factor is applied to the ambient component. Since the algorithm is really simple and does not need any additional data, that is not available already from previous rendering steps. It can be computed in a post-processing step.

When just randomly sampling the area in the close proximity of the pixel, unwanted parts of the scene might be darkened. For example planes, that are almost parallel with the viewing angle, because the depths for about half of the sampling points is smaller then the pixel's depth (parts of the plane closer to the camera). The additional knowledge about the normals enables to generate only samples, that should be in the hemisphere around the pixel (Figure 3.1). Since the screen-space ambient occlusion works only in 2D, the aforementioned hemisphere is only an approximation of the real 3D hemisphere around the point on the surface corresponding to the pixel. How close this approximation is, depends on the right parameters for the algorithm: the size of the sampling area; and the maximal depth difference between the pixel and the sample, where occluders are still considered.

Uniform sampling in global illumination techniques usually causes unwanted artifacts, too. Absolute random distribution of sampling points, however, can also cause problems. Therefore the distribution of the sampling points is required to be close to uniform, but still random. Another problem with random sampling is noise. In some cases the noise can be tolerated, but more often smoothing is used.

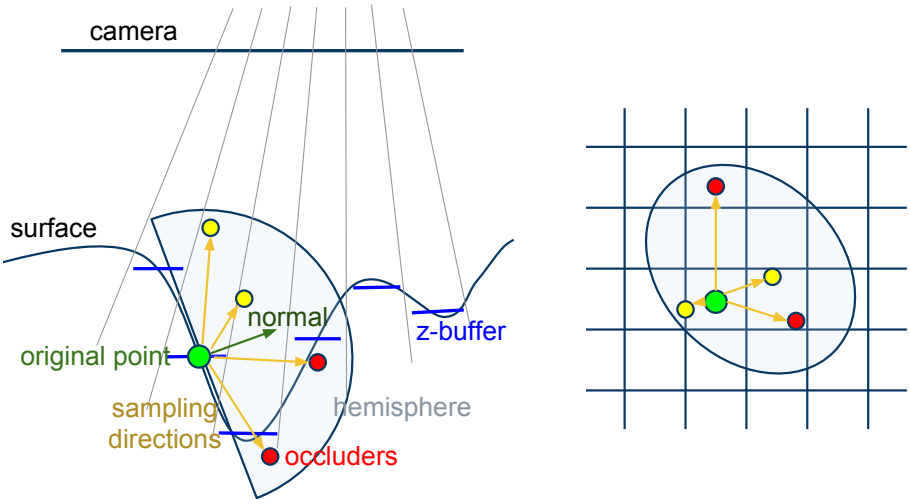


Figure 3.1: Screen-space ambient occlusion. An area around the pixel is sampled in 2D (right part). Based on the normal, sampling points are generated inside the hemisphere (left image). The generated depths for each sampling point are compared to the depth of the appropriate pixel to determine if the pixel corresponds to an occluder object. Yellow (light) points - no occluder; Red (dark) points - occluder. Based on the results of the depth comparisons, the pixel's intensity will be darkened (on the image to half, since half of the sampling points are below scene surface).

The problem of creating a random distribution is frequently solved by precomputing a few uniformly distributed random directions in a sphere and using random normals to reflect these directions. These randomly generated normals ensure some variation of sampling directions across pixels. The reflected directions are then optionally reversed to be in the hemisphere around the normal of the pixel. In this project I used Halton sequences with appropriately chosen bases to get random samples with uniform distribution (Section 3.4).

Computing ambient occlusion from simple depth values has some drawbacks, too. First, the precision of screen-space ambient occlusion is highly dependent on the size of the scene. The bigger the scene, the coarser the object shape approximation. This can lead to unwanted effects, like darkening the whole silhouette of an object.

The second, less frequently occurring problem is caused by the limited area that is sampled for occluders. Let us consider two objects close to each other in the scene in 3D. Using a classical ambient occlusion method, such as using ray tracing, the two objects are darkened. However, using screen-space ambient occlusion the distance between the projected positions of the objects in 2D might be larger then the sampled area. As a consequence screen-space ambient occlusion will not detect any occluders and the objects will not be darkened. This problem emerges mostly for close-up views of objects.

3.3 Screen-space Directional Occlusion

Screen-space directional occlusion is a fast approximation of global illumination. It works in the screen space, takes into account the direction of the light, and is able to handle one indirect bounce of diffuse light [19]. In order to compute light transport, screen-space directional occlusion uses the 3D positions and normals of each pixel in the screen space as

input. The output is created in two passes. The first pass computes the direct illumination. The second pass then uses the data from the previous pass to get indirect bounces of light.

Direct Illumination Using Directional Occlusion

While standard screen-space ambient occlusion methods use only the position and normal of each pixel, screen-space directional occlusion also takes into account the direction of the incoming light. The amount of directional light is computed as follows for each point $\mathbf{P}$ and normal $\mathbf{N}$:

$$L_{dir}(\mathbf{P}) = \frac{1}{\pi} \int_{\Omega} \frac{\rho}{\pi} L_{in}(\omega) V(\omega) (\mathbf{N} \cdot \omega) \, \mathrm{d}\omega, \tag{3.2}$$

where $\frac{\rho}{\pi}$ is the diffuse BRDF, L_{in} is the incoming radiance from direction ω in the hemisphere Ω and V is the visibility test. When using Monte Carlo sampling the integral is replaced by a sum of K samples each covering a solid angle of $\Delta\omega = 2\pi/K$:

$$L_{dir}(\mathbf{P}) = \sum_{i=1}^{K} \frac{\rho}{\pi} L_{in}(\omega_i) V(\omega_i) (\mathbf{N} \cdot \omega_i) \Delta\omega. \tag{3.3}$$

This method assumes that L_{in} can be efficiently computed from environment maps, point or directional lights. Similar to screen-space ambient occlusion, avoiding ray-tracing is done by approximating the occluders in the screen space. The difference is that while in screen-space ambient occlusion the samples are generated in 2D in the image space, screen-space directional occlusion uses sampling points generated in the hemisphere in 3D using the normal and the 3D position of the pixel. The sample points are then backprojected into the image space in order to determine, if in the given direction there is an occluder. This way screen-space directional occlusion does not suffer from the problem mentioned with screen-space ambient occlusion, that is: objects further away in the screen-space, but closer in 3D in the scene may cause occlusion. Another advantage is that the parameters for searching occluders are more straightforward in screen-space directional occlusion. It requires only one parameter, the size of the hemisphere, in comparison with screen-space ambient occlusion, where the area of the 2D area used for sampling and the maximum depth difference is needed.

The sampling of the hemisphere is done in the following way: for every generated direction ω_i and a random step $r_i \in [0 \,..\, r_{max}]$, the position of the sampling points is computed as $\mathbf{P} + r_i\omega_i$. The generated points are located in the hemisphere with a center of $\mathbf{P}$ and oriented around the normal $\mathbf{N}$. The depth of the backprojected sampling points is compared with the values from the original z-buffer. If the depth value from the original z-buffer is smaller than the depth of the sampling point, the sampling point is below the surface. The light from this direction is blocked by an occluder. Otherwise, the light has a clear path from this direction and the incoming radiance can be computed.

The method is demonstrated in Figure 3.2 for four sampling points A, B, C and D (red dots). The sampling points are generated randomly with a uniform distribution over the hemisphere with a random step from the original point, and then backprojected onto the image. Now that the image space coordinates are known, the 3D coordinates can be computed or read from a frame buffer (semi transparent green, orange and white dots). These points are again projected into the image in order to get the distance from the camera. If the sampling point is further than the appropriate point on a surface in the scene, the sampling point is classified as an occluder (A and D points). Otherwise, the illumination

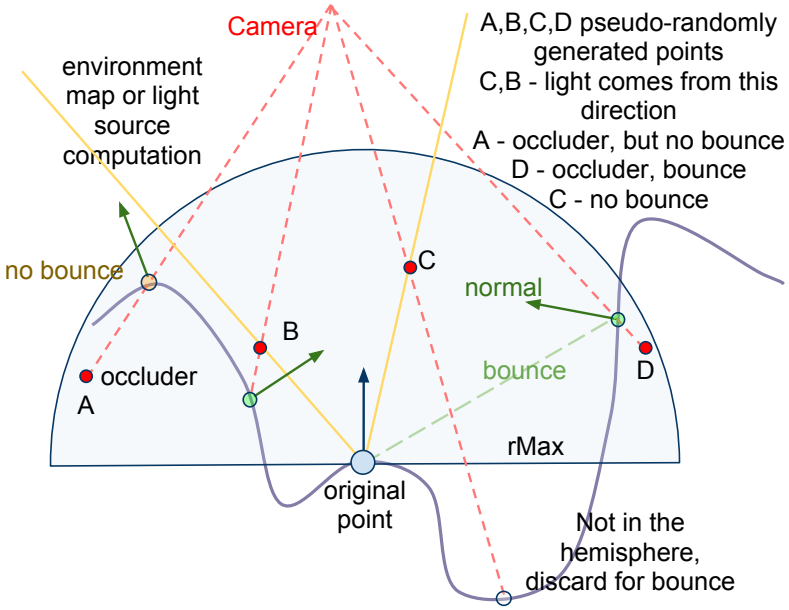


Figure 3.2: Screen-space direcional occlusion principle. Random samples are generated in 3D in the hemisphere. Samples under the surface are classified as occluders. Otherwise, the incoming radiance can be computed from the direction defined by the sampling point. The sampling points classified as occluders are projected on the surface. Based on the color and position of the pixels on the surface, indirect bounces are computed.

can be computed from the direction defined by the point and the origin (B and C points). These directions are shown as yellow lines in the image.

Indirect Bounce

Since the 3D position and normal are available for each sampling point projected to the surface, they can be used to get one indirect bounce of light from the given directions. The method takes into account only the points on the surface projected from sampling points classified as occluders (A and D points) for bounce. For each of these pixels the computed directional light intensity and the corresponding pixel color from the direct illumination pass is used as the base for indirect light. In order to calculate the indirect radiance sent in the direction of the original point, these pixels are treated as small patches oriented around the normal. Using the sender normal, back facing patches can be filtered out (for example, for sampling point A).

The equation for the additional incoming indirect radiance for a point $\mathbf{P}$:

$$L_{ind}(\mathbf{P}) = \frac{1}{\pi} \int_{\Omega} \frac{\rho}{\pi} L_{dir}(\mathbf{P}_\omega)(1 - V(\omega)) \cdot \frac{A_s(\mathbf{N} \cdot \omega)(\mathbf{N}_\omega \cdot (-\omega))}{|\mathbf{P} - \mathbf{P}_\omega|^2} \, \mathrm{d}\omega,$$

where $\mathbf{P}_\omega$ and $\mathbf{N}_\omega$ are the point and normal from point and normal buffer, each corresponding to a sampling point taken from the hemisphere in direction ω. A_s is the area associated with the sender patch. This equation respects the mutual orientation of the surfaces based on the normals ($(\mathbf{N}\cdot\omega)(\mathbf{N}_\omega\cdot(-\omega))$) and that the intensity of the incoming radiance decreases quadratically with growing distance between the surfaces.

The modified version of the equations for K samples is then:

$$L_{ind}(\mathbf{P}) = \sum_{i=1}^{K} \frac{\rho}{\pi} L_i(1 - V(\omega_i)) \frac{A_s(\mathbf{N} \cdot \omega_i)(\mathbf{N}_i \cdot (-\omega_i))}{|\mathbf{P} - \mathbf{P}_i|^2} \Delta\omega, \tag{3.4}$$

For the initial value for A_s, the base circle is subdivided into K regions, each covering $A_s = \pi r_{max}^2 / K$. This value can also be used as a parameter to control the strength of the color bleeding manually.

In the example above (Figure 3.2), points A and D are the only occluders. After projecting these points onto the surface from the camera's viewpoint, the information about the normal, position and color are also available. The projected point for A has a back facing normal, consequently it will not contribute to the final bounce. The patch for sampling point D, on the other hand, will qualify as a sender of indirect light towards point $\mathbf{P}$.

3.4 Halton Sequence

With both screen-space ambient and directional occlusion I used Monte Carlo sampling to approximate the correct solutions. Absolute random distribution of samples in Monte Carlo methods can cause problems; one of the worst of these is clumping (when for small number of random numbers, the variance between the values is low). The effect of clumping in quasi-Monte Carlo methods is decreased by eliminating the randomness of the generated values completely. Samples are deterministically computed to achieve a stochastic distribution as close to the uniform distribution as possible.

In order to describe how much the point distribution of a given method derivates from an ideal solution, a measure called discrepancy is used. Quasi-Monte Carlo methods try to minimize this discrepancy. There are several low-discrepancy sequences that are used for generating sampling points: Hammersley, Halton, Sobol, Niederreiter, etc. [9]

The Halton sequence generation is based on the radical inverse function applied to an integer i. This integer can be expressed in a base b with terms a_j:

$$i = \sum_{j=0}^{\infty} a_j(i) b^j. \tag{3.5}$$

The radical inverse function is computed by reflecting the resulting digit sequence around the decimal point:

$$\Phi_b(i) = \sum_{j=0}^{\infty} a_j(i) b^{-j-1}. \tag{3.6}$$

Examples of the radical inverse function for numbers 1 to 6 in base 2 and 3 are shown in Table 3.1.

For generating multi-dimensional low-discrepancy sequences, a different radical-inverse sequence is used in each dimension. The ith point in the sequence is given as:

$$x_i = (\Phi_{b_1}(i), \Phi_{b_2}(i), \ldots, \Phi_{b_d}(i)), \tag{3.7}$$

where the bases b_j are relatively prime[1] and d is the dimension of the sequence.

[1] relatively prime numbers have no common positive divisor other then 1; in other words their greatest common divisor is 1

An intuitive explanation of the uniformness of the Halton sequence is as follows. Let us consider the generated floating point numbers as digit sequences in the given base expressed as strings. Before generating strings of length $m+1$, all the strings of length m are produced. This means that before generating a new point on an interval, all intervals of size $b-m$ will be visited first. This fact also suggests some kind of periodicity in similarity of the generated values. For example, let us consider a Halton sequence with base 2: $\Phi_2(50) = 0.296875, \Phi_2(50+16) = 0.2578125, \Phi_2(50+32) = 0.2890625, \Phi_2(50+64) = 0.3046875$. This periodicity can be expressed [5]:

$$|(\Phi_b(i) - \Phi_b(i + mN_g)| < \frac{1}{b^k}; N_g = lb^k, l > 0, k \geq 0, \tag{3.8}$$

where l, m, k are integers, and N_g is the period to generate similar sampling points. For multidimensional Halton sequences, the least common multiple of the periods for each dimension is used as N_g.

3.5 Bilateral Filtering

Bilateral filtering was developed by Tomasi and Manduchi [22] as an alternative to anisotropic filtering. Bilateral filter is a non-separable non-linear filter, where the output is the weighted average of the input.

The original version of the bilateral filtering works as follows. The filter starts with a standard Gaussian filter with a spatial kernel c. However, in order to avoid blurring over edges, the weight of each point depends also on a function g in the intensity domain. This function decreases the weight of points in the image with large intensity differences from the intensity of the currently filtered point. For this property, the g function is also referred to as the edge stopping function. The output of the bilateral filter for a grayscale image $f(\mathbf{x})$:

$$h(\mathbf{x}) = \frac{1}{k(\mathbf{x})} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c(\xi, \mathbf{x}) g(f(\xi), f(\mathbf{x})) f(\xi) \, \mathrm{d}\xi, \tag{3.9}$$

where $k(\mathbf{x})$ is the normalization term:

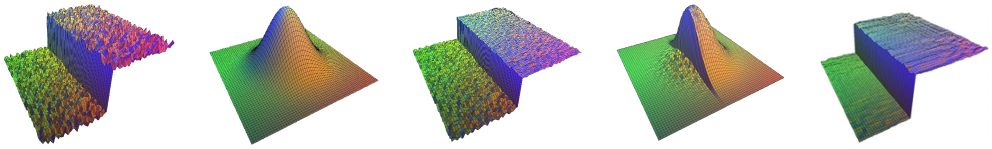
$$k(\mathbf{x}) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c(\xi, \mathbf{x}) g(f(\xi), f(\mathbf{x})) \, \mathrm{d}\xi, \tag{3.10}$$

The effect of the bilateral filter is also demonstrated on Figure 3.3.

In practice the bilateral filtering is usually implemented with a Gaussian for function c in spatial space and a Gaussian in the intensity domain for function g. As a consequence the value at point x is only influenced by points that are spatially close and have similar

i	Reflection for base 2	Reflection for base 3	$\Phi_2(i)$	$\Phi_3(i)$
$1 = 1_2 = 1_3$	$.1_2 = 1/2$	$.1_3 = 1/3$	0.5	$0.3\overline{3}$
$2 = 10_2 = 2_3$	$.01_2 = 1/4$	$.2_3 = 2/3$	0.25	$0.6\overline{6}$
$3 = 11_2 = 10_3$	$.11_2 = 1/2 + 1/4$	$.01_3 = 1/9$	0.75	$0.1\overline{1}$
$4 = 100_2 = 11_3$	$.001_2 = 1/8$	$.11_3 = 1/3 + 1/9$	0.125	$0.4\overline{4}$
$5 = 101_2 = 12_3$	$.101_2 = 1/2 + 1/8$	$.21_3 = 2/3 + 1/9$	0.625	$0.7\overline{7}$
$6 = 110_2 = 20_3$	$.011_2 = 1/4 + 1/8$	$.02_3 = 2/9$	0.375	$0.2\overline{2}$

Table 3.1: Examples of radical inverse function for bases 2 and 3



(a) input (b) spatial Gaussian c (c) influence of g edge stopping function (d) resulting weight $k(\mathbf{x})$ (e) output

Figure 3.3: Bilateral filtering (colors are used only to convey shape). [7]

intensities. Taking advantage of this, the integral in real applications is replaced by a sum over a kernel with defined size:

$$J_p = \frac{1}{k_p} \sum_{q \in \Omega} I_q c(p-q) g(I_p - I_q), \tag{3.11}$$

where I_p, I_q are intensities from the original image, Ω is the spatial support for the kernel c, and k_p is the normalization factor.

Geometry-aware Filter

Bilateral filtering can be generalized to use different kinds of edge stopping functions, not just the function based on the intensities for each point. In order to blur intensities respecting geometric properties of the scene, the bilateral kernel can be defined over [17]:

- Domain: a spatial Gaussian kernel like in the original bilateral filter.
- Range: a combination of kernels over distance in object space or depth values, angle between the normals and optionally values that are to be filtered.

A filter defined this way would be geometry-aware in the sense, that it will not blur intensities between different objects and over corners and edges of the same object. However, on highly complex scenes high frequency details of intensities might get lost.

Joint Bilateral Upsampling

The idea behind the bilateral filter can also be applied for upsampling. This approach is called joint bilateral upsampling [13]. Joint bilateral upsampling assumes, that additional information for filtering is available in the original high-resolution, while the computed solution is in a lower resoluion:

$$\overline{S_p} = \frac{1}{k_p} \sum_{q_\downarrow \in \Omega} S_{q_\downarrow} c(\|p_\downarrow - q_\downarrow\|) g(\|\overline{I_p} - \overline{I_q}\|), \tag{3.12}$$

where $\downarrow$ represents the downsampled and $\overline{\square}$ the information in higher resolution; S is the computed low resolution solutions; $p_\downarrow, q_\downarrow$ denote the corresponding downsampled coordinates; $\overline{I_p}, \overline{I_q}$ are the information from the high-resolution image.

In order to get scene-geometry based upsampling, the same modifications can be applied as in the case of the bilateral filtering. The information about normal and depth in higher resolutions could be used for the g function.

Chapter 4

Proposed Improvements to Screen-space Directional Occlusion

The way how the standard screen-space directional occlusion method calculates lighting has some disadvantages. Firstly, for every pixel the same number of sampling points is generated. This might be a waste of computing power; especially when there are great planes in the scene. Occlusion only happens at places, where the geometry is slightly more complicated. It would be beneficial to generate more sampling points for areas, where it is worth it, and less for regions in the scene, where only a small number of sampling points is necessary to get fairly good results (Section 4.1).

Another potential problem could be that screen-space directional occlusion computes lighting for every sample. Due to random sampling, this fact could cause noise even on plane surfaces for smaller number of sampling points. One can try to smooth the results (Section 4.3). However smoothing intensities might cause loosing some high frequency details. When computing illumination from an environment map this is unfortunately unavoidable, but for point and directional lights instead of intensity for each pixel a darkening factor could be computed (Section 4.2).

4.1 Sampling

Both screen-space ambient occlusion and screen-space directional occlusion are in a way trying to approximate ambient occlusion. This approximation does not just mean approximating the results in screen space, but also using much less sampling directions. Covering every angle in the hemisphere and doing a visibility test for each direction would require too much computing power.

Generating sampling points in itself is a quite costly operation. Most applications trade speed of the sampling over the quality of the sampling. For example, a solution could be storing a small number of constant directions in a sphere. When doing the sampling, these directions are optionally reversed to be in the hemisphere. In order to add some variance into the directions, a random direction is generated, which is used to reflect the stored constant directions.

This approach in the one hand can be really fast, on the other hand, it does not allow generating more samples then the number of stored directions and guaranteeing a finer distribution of the sampling directions. Lets consider 8 constant directions in the sphere. For generating $16, 24, ...$ directions, $2, 3, ...$ random directions are needed. Random direc-

tions, however, may cause clumping. Some of the directions could be very close, hence not improving the overall result.

In this project I used Halton sequences to generate directions in the hemisphere. This way I was able to generate more sampling directions and get finer and finer coverage of the hemisphere. I stored the Halton sequences in textures. Unfortunately the disadvantage of this method is that it requires a texture lookup for every new direction. On the other hand the values are read sequentially from the texture, so caching values could improve speed substantially.

Halton Sequence Periodicity

As mentioned in Section 3.4 Halton sequences have a periodic property. For example lets consider two Halton sequences with bases 2 and 5. The possible periods for Halton sequence with base 2 based on equation (3.8) are: $2, 4, 6, 8, ..., k \cdot 2^l$; for the Halton sequence with base 5: $5, 10, 15, 20, 25, ..., s \cdot 5^t$. The possible periods for the combination of these sequences, where both sequence's period effect the result of splitting the hemisphere, are $10, 20, 40, 50, 80, ...$ etc. This is demonstrated in Figure 4.1

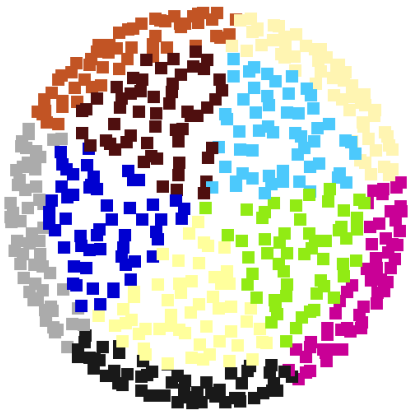


Figure 4.1: Projected view of the hemisphere - Halton sequences with bases 2, 5 and period 10. Each pixel is colored based on the index in the sequence and the period. Every 10th pixel has the same color.

Note, that equation (3.8) enables to use multiplies of the powers of the bases as periods. But even though the directions are similar, they cannot be used to split the intervals into multiple parts. For example for period $3 \cdot 2^1$ and 5, it has the same effect regarding the splitting the hemisphere as using 2 and 5 as periods (See Figure 4.2).

The aforementioned periodicity property can be exploited to control the number of sampling points and still have a quasi-uniform distribution. For example, with Halton sequences with period 10, 10 samples can be used for pixels where the low number of sampling points does not matter. For pixels where more sampling points are needed to get smoother results, $20, 30, ..., k \cdot 10$; $k \in N$ sampling points can be used.

Another possibility for future improvements exploiting periodicity might be, for example, in the case of occlusion detection to filter out directions, where an occluder was already found. The samples could be generated with gradually growing distance from the original point. This would allow using larger hemisphere for sampling and still have good results around smaller objects.

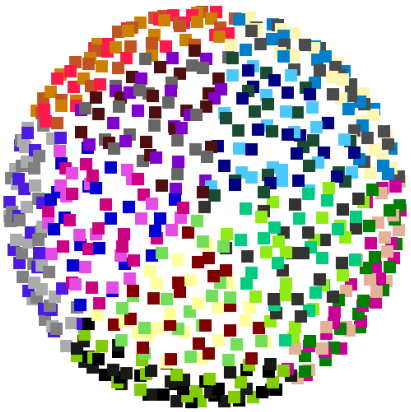
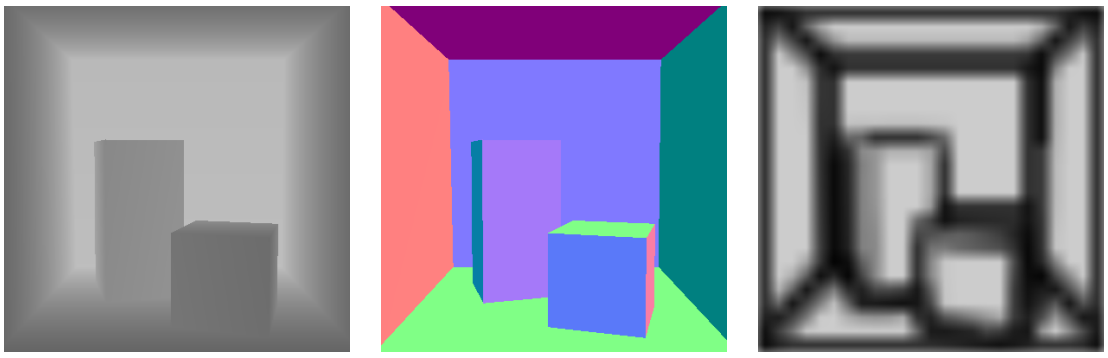


Figure 4.2: Projected view of the hemisphere - Halton sequences with bases 6, 5 and period 30. Each pixel is colored based on the index in the sequence and the period. Every 30th pixel has the same color.

Preprocessing

As mentioned earlier in standard screen-space directional occlusion, the same amount of sampling points is generated for every pixel. Setting this amount higher means better results, but it will cause a performance drop. Generating more samples for planes, which do not have any occluder nearby, is a waste of computing power. The solution would be to generate more sampling points for pixels which are potentially occluded and less for pixels where the probability of finding an occluder is low.

The number of occluders is usually higher at parts of the image where the normals or the depth values differ significantly. In order to get areas where higher number of sampling points should be generated, a preprocessing step could be added to screen-space directional occlusion calculations. In this preprocessing step a weight is calculated to define the number of sampling points to be used. Areas where the changes in normal and depth values are bigger, are given a higher weight. Planes, on the other hand, will have much smaller weight since the normals are constant for every pixel on the same plane.



(a) depths (b) normals (c) weight

Figure 4.3: The computed weight for preprocessing step. The darker the higher the weight.

The weight is calculated using a simple edge detection filter using the normals and depth values. The calculated weight then represents the gradient magnitude of the normals and

depths (Figure 4.3). This value can be used to determine the number of sampling points. In order to get wider areas around discontinuities and also to speed up the filtering, it can be computed in much lower resolutions.

Using just one resolution for filtering is not the best solution. Regions of interest also depend on how close the camera is to the objects. Combining results from different resolutions should give more precise results of areas, with higher possibility of finding occluders. A more advanced method was for example used by Nichols et al. [15] to get locations where higher resolution was required for computing image space radiosity.

4.2 Point and Directional Lights

Screen-space directional occlusion can compute incoming radiance for each pixel either from point and directional lights or environment maps. When using only point and directional lights there are a few modifications possible. As mentioned in the introduction for this chapter, computing intensities for each sample has its disadvantages. Noise can appear even on plain surfaces due to random sampling. This noise unfortunately cannot be smoothed excessively, because high frequencies of intensity would disappear, thus more sampling points would be needed to get smoother results.

In order to circumvent this problem, I computed screen-space directional occlusion in the following way: when rendering the scene before computing screen-space directional occlusion to get the material, normal and depth information, the overall intensity of the pixels can be computed using the appropriate lighting model. The overall intensity should be equal to the value computed using equation (3.3) without the visibility function.

$$L_{overall}(\mathbf{P}) = \frac{1}{\pi} \int_{\Omega} \frac{\rho}{\pi} L_{in}(\omega)(\mathbf{N} \cdot \omega) \, \mathrm{d}\omega, \tag{4.1}$$

When computing the direct illumination step in screen-space directional occlusion, instead of computing the intensity for each pixel, a modulation factor can be calculated, similar to ambient occlusion. This modulation factor is the ratio of the intensity computed with the visibility function and without it:

$$M(\mathbf{P}) = \frac{L_{dir}(\mathbf{P})}{L_{overall}(\mathbf{P})} \tag{4.2}$$

$$= \frac{\int_{\Omega} \frac{\rho}{\pi} L_{in}(\omega) V(\omega)(\mathbf{N} \cdot \omega) \, \mathrm{d}\omega}{\int_{\Omega} \frac{\rho}{\pi} L_{in}(\omega)(\mathbf{N} \cdot \omega) \, \mathrm{d}\omega}. \tag{4.3}$$

For K samples after simplifying the equation:

$$M(\mathbf{P}) = \frac{\sum_{i=1}^{K} L_{in}(\omega_i) V(\omega_i)(\mathbf{N} \cdot \omega_i)}{\sum_{i=1}^{K} L_{in}(\omega_i)(\mathbf{N} \cdot \omega_i)} \tag{4.4}$$

The modulation factor can either be per channel or a single value based on the lighting properties. For scenes with multiple lights with radically different colors, a darkening factor for each color channel is best. However, for most scenes a single modulation factor is sufficient, since the lights are usually white or the distance between them is large enough, so they have only a minor effect on objects close to other lights. A single modulation factor also helps to reduce memory usage.

This modulation factor can further be processed. Unlike smoothing intensities, smoothing the modulation factor does not cause loss of high frequency intensity details in the

original scene. Since it is actually a generalized version of the darkening factor computed with ambient occlusion, it changes, similar to ambient occlusion factor, slowly on plain surfaces. This permits one to compute it in lower resolution and also to use larger smoothing kernels. As a consequence the method could be sped up radically, and also the noise can be reduced significantly.

More speed improvements can be achieved by merging the two steps of screen-space direction occlusion together. This means that the source color for the bounces has to be taken from the scene rendered without the screen-space directional occlusion darkening. In order to avoid pixels being too bright in dim corners, the indirect bounces should also be darkened using the modulation factor. Thanks to computing the darkening factor and bounces in one step, the sampling points are generated only once. Another advantage of merging the two steps is that the modulation factor and bounce can be optionally blurred together in one step.

4.3 Smoothing

Random sampling is used in Monte Carlo methods to reduce the effect of artifacts, but random sampling, even quasi-random sampling causes noise in the final image. Unless the noise is relatively small, it is needed to be smoothed. In the case of using Halton sequences the need for filtering might even be higher, because it is deterministically computed. Also they have a periodic property, so recurring patterns, parallel lines might appear in the noise.

In order to suppress noise and artifacts geometry-aware blurring can be used. A simple Gaussian blur is not enough in this case. The usage of geometric information is necessary to prevent bleeding of intensities and darkening values over edges and between distant pixels in 3D, but close in 2D. A geometry-aware filtering, like a modified bilateral filtering on the base of normals and depth values described in Section 3.5 is appropriate.

The downside of bilateral filtering is that it is inseparable and thus requires many texture lookups. The performance drop caused by a full bilateral filter would not be worth the smoothing. So as to approximate the results of a full bilateral filter, Reinbothe et al. [17] separated the calculations into a vertical and horizontal pass. The combination of the results of these two one-dimensional filters should improve the frame rates significantly and still provides an acceptable quality.

Even with separated bilateral filtering, if small number of sampling points is used and the kernel size for blurring is set too high, the speed of the method might be slower, than using more sampling points and smaller smoothing kernel. Also with excessive blurring the approximation of the full bilateral filtering gets coarser and artifacts might appear.

4.4 Subsampling

The original two step screen-space directional occlusion method does not enable subsampling, since it computes intensities for pixels directly. Subsampling would cause loss of high frequency data. However, indirect bounces and the modulation factor from Section 4.2, can be both subsampled. This is possible given the assumption, that indirect lighting and ambient occlusion values change slowly over spatial space.

Subsampling helps to speed up the methods radically. The number of pixels, that have to be processed in the fragment shaders drops quadratically with the magnitude of subsampling. Subsampling does not only help the screen-space directional occlusion computations.

The smoothing is also done in lower resolutions. Hence, fewer pixels have to be processed to get the same blurring effect than in full resolution.

A disadvantage of subsampling is that the computed values have to be upsampled correctly. To do this the geometry properties of the scene have to be taken into account. For upsampling I used a modified joint-bilateral filtering described in Section 3.5. Upsampling the values this way also adds some additional blur to the modulation factor and indirect bounces.

Even though subsampling does help the speed, and a geometry aware upsampling does a relatively good job not adding any disturbing artifacts, too much subsampling would mean loss of finer details. Also the number of sampling points for screen-space directional occlusion have to be increased, otherwise plain surfaces could become spotty even with smoothing enabled (low frequency noise).

Chapter 5

Implementation Design

When implementing 3D graphical applications, one always has to consider using already existing game engines and other utilities. They usually have many repetitive tasks already implemented and help the programmer to concentrate on the given task. On the other hand they restrict the programmer to use certain tools. Also they might not have facilities that would be essential for implementing the current problem. I wanted the application to be cross platform, using open standards and as simple as possible, leaving only the bare minimum of functionality that is useful for demonstrating the implemented methods.

I decided to use OpenGL version 3.2 and over, since it supports all the needed functionality in the core:

1. multiple render targets (aka. rendering to textures) - the most important feature;
2. vertex buffer objects, vertex array objects - for efficient drawing;
3. floating point textures - they are important for high dynamic range rendering and to get more precise results for global illumination and variance shadow mapping;
4. advanced shaders.

There are not many engines based on OpenGL3, so this already reduces the number of candidates. Most game engines still rely on OpenGL 2.1 or compatibility mode in newer version of OpenGL with the combination of the extensions. For example the glux engine from GluxMark [23] does meet all these criteria, but I found it difficult to add more post-processing steps. Since both screen-space ambient occlusion and screen-space directional occlusion rely on this, I rather chose a different solution.

Because I could not find any engine that would be easy to use and meet every criterion, I decided to write my application from scratch. This gave me the additional freedom to implement only the features I wanted and features that were needed to demonstrate the methods. Additionally it has the advantage of supplying results of the methods raw speed, without any external influence. The downside of creating the application this way, that the implementation is not bond to be the most optimized and to use the best solution in places for a given problem.

Almost every method I wanted to implement relied on modifying the results of previous steps in a post-processing step. Screen-space ambient occlusion and the modified screen-space directional occlusion compute the darkening factor and the bounce base on the the normal, depth and color information from the direct rendering step. Bilateral smoothing

processes the bounce and modulation factor from screen-space ambient and directional occlusion. High dynamic range rendering needs a post-processing step to scale intensities and apply the tone-mapping operator.

In order to be be able to dynamically enable/disable each method, the application needed to have an abstraction that could handle each step identically. To achieve this I separated the rendering of the scene into steps:

1. Shadow mapping step - step to compute the shadow maps.

2. Direct illumination step - computes the intensity for each pixel and stores the depth, material and normals for each pixel for later use.

3. Global illumination step - computes the darkening factor and bounce for screen-space ambient and directional occlusion; smoothes the bounce and darkening factor, and applies the effects. Optionally upsampling is also done in this step, too.

4. High dynamic range rendering step - the tone mapping operator is applied here.

5. Text rendering step - shows additional information of settings and the current FPS.

Separating the rendering into these steps makes methods belonging to the appropriate rendering step interchangeable without too many complications.

To implement these methods I chose the programming language C++. C++ is an object oriented language which nicely fits the architecture I just described. Every step could be represented by a class implementing a common interface, so that they can be accessed as one. In C++ this is possible via creating a virtual base class, that defines the interface, and deriving all rendering steps from this class. Another advantage of C++, that there are many libraries available to use, and the binaries generated by the compiler usually do not stress the CPU as much, as an alternative implementation would in Java or Python[1].

In order to exploit hardware acceleration I used the OpenGL API version 3.3 and GLSL 150.[2] A simplified chart of the OpenGL3 rendering pipeline is on Figure 5.1[3]. From the programmable stages only the vertex shader and fragment shader stages are important from the application view.

Vertex shader stage transforms the input attributes (vertexes, normals, texture coordinates), and passes it to the next step. The input attributes are stored in vertex buffer objects. Vertex buffer objects can practically contain any type of data, not just the vertex coordinates, normals and texture coordinates. Some effects, for example bump mapping, might also require tangent space coordinates.

Vertex array objects work as a glue. They combine the various input attributes stored in vertex buffer objects. While drawing, the combined data are fed into the OpenGL pipeline based on an element array bound to the vertex array object. The element array is essentially a vertex buffer object containing the indexes into other vertex buffer objects. This is just an simplified explanation how vertex array objects and vertex buffer objects were used in this project for drawing. For more detailed information about vertex array objects and vertex buffer objects and how they can be used, please refer to the OpenGL reference[4].

[1] These are just examples. I considered only object oriented languages

[2] There is already a newer version of the API available (version 4.0), but the graphics card I used for testing did not support it.

[3] http://images.anandtech.com/reviews/video/dx11/dx10pipeline.png

[4] http://www.opengl.org

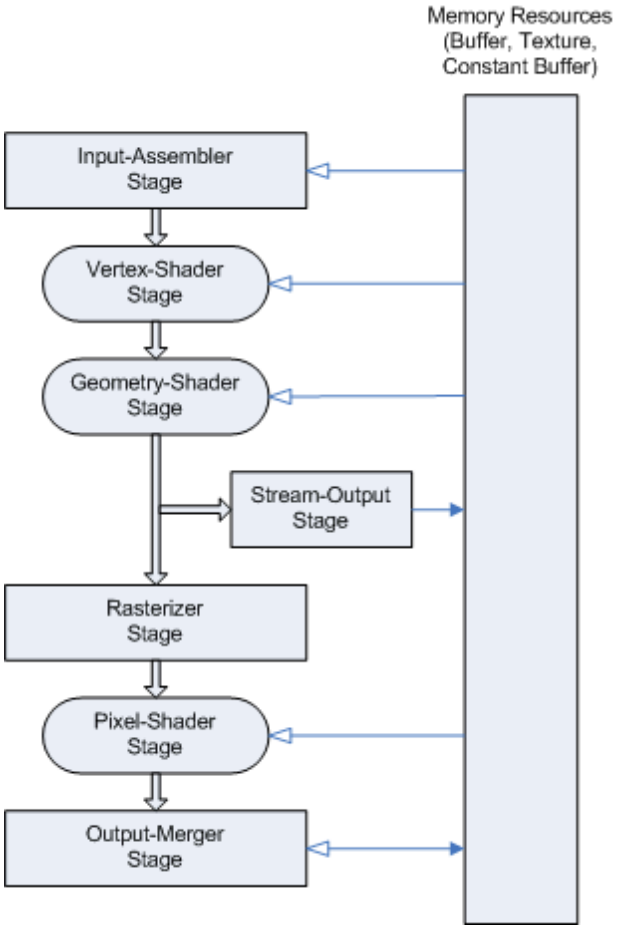


Figure 5.1: OpenGL3 rendering pipeline. (Basically the same as the DirectX 10 pipeline.)

For storing the scene data (vertex data, normals, texture coordinates, material information) outside the application I decided to use the OBJ[5] file format. It is a human readable format first developed by Wavefront Technologies. Compared to other file formats OBJ files require noticeably more data storage, but thanks to its simple syntax, it is easy to parse and extend (for example adding camera, light positions). The material descriptions for objects in OBJ file format are stored in MTL[6] files. Similarly to OBJ, MTL is human readable and easy to parse. Since the OBJ file format is relatively widespread, most 3D modeling applications support exporting to this format. So it was easy to acquire testing scenes for the application.

From the applications perspective much more important stage is the fragment shader stage. Fragment shaders are used for per-pixel lighting, in post-processing steps to compute global illumination methods, high dynamic range rendering and smoothing. The output of the fragment shaders are, for the most part, stored in textures; except for the last step, where the output is rendered directly to the screen's framebuffer. This is possible thanks to the new feature of OpenGL3: framebuffer objects and rendering to textures. Compared to the old OpenGL specification, the aforementioned features also enable to render multiple render targets. This essentially means that during the rendering of the scene from the camera's

[5]http://www.martinreddy.net/gfx/3d/OBJ.spec
[6]http://paulbourke.net/dataformats/mtl/

point of view the normals, the depth values, the pixel colors, materials information, etc., can all be saved into textures in one pass.

In order to be able to use OpenGL, it has to be initialized first by creating a context. Because creating a context requires platform specific extensions, the application uses SDL[7] to handle this. SDL (Simple DirectMedia Layer) claims to be a cross-platform multimedia library. SDL among others can handle initializing OpenGL, create the framebuffer visible to the user, and handle platform specific window managers and process keyboard and mouse input. I also used in the application SDL_image[8] to load images and SDL_ttf[9] for loading and rendering fonts. SDL is a fairly powerful tool that lifts many repetitive tasks from the programmer when creating a 3D application using OpenGL. For this reason SDL enables the programmer to focus on the problems at hand, which were realistic lighting methods in my case.

A minor flow in the architecture separating the rendering into steps in the application as described above is that it doesn't scale well for multiple lights. Initially I used only one light, which is simple to implement and is enough to show the capabilities of the methods. Using only one light is still enough to be able to give a relatively good idea of the differences between screen-space ambient and directional occlusion. The application was designed to do exactly that. However, for comprehensive comparison of the two methods, multiple lights are important. Since I calculated only with a small number of lights, I didn't use deferred shading for computing intensities. Also the screen-space directional and ambient occlusion already stress the texturing unit on the GPU, deferred shading might have slowed down the methods even more.

5.1 Shadow Mapping

To get shadows caused by directional and point lights I implemented three shadow mapping techniques: simple shadow mapping, shadow mapping with percentage closer filtering and variance shadow mapping. In the first step of the rendering the application creates the shadow maps for every light. The shadow maps are then used to compute shadows in the direct illumination step.

Percentage closer filtering and simple shadow mapping only require a depth value for each pixel from the light's point of view. Thus the shadow map generation is really fast. It depends only on the scene geometry, and does not use any texture or normal information. To store the shadow map, the application uses a framebuffer object with a depth texture attached to it.

The shadow map is used in the next step to determine if an object is in shadow or lit. The comparison of depth values can be turned on on the hardware for the depth texture using the `GL_COMPARE_R_TO_TEXTURE` texture mode for OpenGL. In simple shadow mapping only one comparison is made. The percentage closer filtering compares the current depth to depth values around the pixel in the depth texture, too. I used a small kernel to do this. The results of the comparisons are then simply averaged together. In order to get smoother results I also enabled bilinear filtering for shadow maps.

Variance shadow mapping requires two values to be stored: the depth and the square of the depth. In order to get reasonably precise results, these values are stored into a

[7] http://www.libsdl.org
[8] http://www.libsdl.org/projects/SDL_image/
[9] http://www.libsdl.org/projects/SDL_ttf/

floating point RGBA texture. Afterwards the texture is smoothed with Gaussian blur in order to remove jagged edges and to get an area of penumbra. In the blurred texture the blurred depth represents the local mean of the depth values and the local variance σ^2 can be computed the following way:

$$\sigma^2_{depth} = \mu_{depth^2} - \mu^2_{depth} \tag{5.1}$$

Using the variance, the mean and the Chebychev's inequality described in Section 2.1, the fragment shader computes the probability of the pixel being in shadow.

The Gaussian blur for the variance shadow map is done in two steps. The first step blurs the texture in the vertical direction and renders the result into a temporary texture. The second step uses the temporary textures for blurring in the horizontal direction, and renders the results back to the original texture.

5.2 Direct Illumination

The direct illumination step renders the scene from the camera's perspective. This step takes advantage of framebuffer objects to render multiple targets: the intensities, normals, depth and materials. All of these properties are stored in textures for each pixel.

The intensities for each pixel are computed using Phong shading. The material and light source information is passed to the fragment shader using uniforms. This limits the number of possible light sources, but as explained earlier, for demonstration purposes small amount of lights is sufficient. The intensities also depend on the result of shadow mapping. (For description of shadow computations see previous section.)

The normals are stored in view space into an RGBA texture. View space normals help screen-space ambient occlusion to easier filter out points not in the hemisphere. (The normal's x,y and the view-space x,y coordinates grow in the same direction.) The alpha channel in the texture is used to store linear depth values. These are later used for bilateral filtering and screen-space directional occlusion.

Since the linear depth values are not as precise as the non-linear depth buffer computed by OpenGL, for screen-space directional occlusion the latter is more suitable. When back-projecting the pixels into 3D space from the non-linear depth buffer, the precision for near objects is much higher. For objects further away from the camera the lower precision does not matter, because screen-space directional occlusion works based on the available visible screen-space data, and the fine details for distant objects cannot be seen in the rendered images anyways.

5.3 Screen-space Directional Occlusion

In order to get a reference for the results of the proposed improvements for the screen-space directional occlusion, I decided to implement the original two step algorithm, too. The full description of the algorithm is in Section 3.3. In the first step it uses the depth, normals, materials from textures from the previous step and light positions to compute direct lighting. The results are again rendered into a texture. This texture is used in the second step to compute bounces. The only main difference in implementation compared to the method with proposed improvements is that in the latter the bounce is computed in the same shader to avoid sampling twice. So the implementation design described below, except the improvements, was the same for both versions, unless specified otherwise.

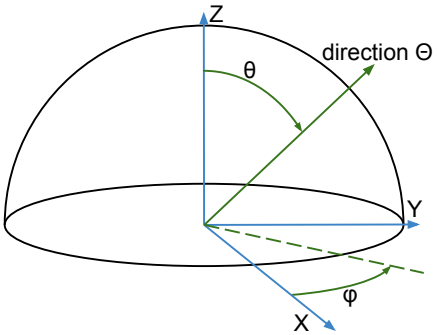


Figure 5.2: The angles for a direction in the hemisphere.

For sampling I used three Halton sequences with bases 2, 5 and 3 (Section 4.1). The first two sequences are used to determine direction and the third is used as the step size from the original point. Let us consider an orthogonal coordinate system around the pixel's normal with the z axis parallel with the normal. A random direction proportional to cosine-weighted solid angle is computed the following way [8]:

$$\begin{aligned} x &= \cos(2\pi r_1)\sqrt{1-r_2}, \\ y &= \sin(2\pi r_1)\sqrt{1-r_2}, \\ z &= \sqrt{r_2}. \end{aligned} \tag{5.2}$$

This gives a probability of $p(\Theta) = \frac{cos\theta}{\pi}$ for direction Θ defined by angles: $\theta = cos^{-1}\sqrt{r_2}, \theta \in [0, \pi]$ (angle between the direction and the z axis) and $\phi = 2\pi r_1, \phi \in [0, 2\pi]$ (the rotation angle around the z axis, Figure 5.2). For random number r_1 from the Halton sequence with base 5 and r_2 from the Halton sequence with base 2. Generating samples this way with period 10, splits the hemisphere vertically into two parts and around the normal into five (Figure 4.1).

For the original screen-space directional occlusion the application uses directions with probability proportional to solid angle. This is necessary because the intensities are computed for each sampling point, and using a cosine-weighted probability surfaces with the normal parallel with the light direction would be brighter and surfaces with the normal perpendicular with the light direction darker. The direction is computed similarly:

$$\begin{aligned} x &= \cos(2\pi r_1)\sqrt{1-r_2^2}, \\ y &= \sin(2\pi r_1)\sqrt{1-r_2^2}, \\ z &= r_2. \end{aligned} \tag{5.3}$$

In order to exploit the periodicity of the Halton sequences as described in Section 4.1, the preprocessing step is computed first for screen-space directional occlusion. This can be done simply with using only the normal and linear depth values from the direct illumination step. The fragment shader can use a small kernel for this step in a lower resolution. This value is then read from the main step of the screen-space directional occlusion.

In the main step of the screen-space directional occlusion the shader loops through all the samples, and computes the incoming intensity from lights with and without the occlusion function (the detailed algorithm is in Section 3.3 and Section 4.2). As mentioned earlier, in

the modified version computing the bounce and modulation factor is merged, hence bouncing is computed in the same loop over the samples in the shader.

Similar to every step, the results are rendered into textures. Afterwards these textures are filtered using the separated bilateral filtering described in Section **??**. I decided to compute the modulation factor for every channel to have a direct comparison with the original screen-space directional occlusion. This implies that the modulation factor and bounce are stored in separate textures, and are smoothed separately. If the modulation factor was just single value, it could have been stored as the alpha channel in the bounce texture. That would have meant less memory usage and only one texture to smooth. In the future this could represent another way to approximate and speed up the method.

In the end the smoothed bounce and modulation factor is applied to the intensities for each pixel. But before this could be done, if subsampling was used, the values are upsampled using joint-bilateral upsampling (Section 3.5) using the normal and the linear-depth for the edge stopping function. I decided not to separate the bilateral upsampling into two steps in order to get better results for edges. Instead, I used a fairly small kernel for upsampling (3x3 pixels). Small kernel also limits the magnitude of the subsampling, because after a certain point artifacts might appear on the edges. However, since too much subsampling means loss of detail, which is undesirable after a certain point, the small kernel should not be a major problem.

After the values have been correctly upsampled, for each pixel the bounce and the intensity is then darkened using the modulation factor, then summed together. The results are copied back into the intensity texture. The user can control a two parameters for screen-space directional occlusion: the bounce strength and the radius for sampling. Compared to screen-space ambient occlusion adjusting these values is fairly intuitive.

5.4 Screen-space Ambient Occlusion

Screen-space ambient occlusion is fairly easy to implement. One of the simplest implementations would be to sample the sphere around the point. In order to avoid dark planes the modulation factor is corrected by 0.5 given that the modulation factor was computed on the interval $(0, 1)$. This basically makes half of the samples redundant.

I decided to use the normal besides the depth, so that all the samples generated are in the hemisphere around the point. I also used the same algorithm for sampling than for screen-space directional occlusion in order to eliminate the speed difference caused by different type of sampling between the two methods. (For pros and cons for sampling using the Halton sequence see Section 4.1). First, a direction is generated using the Halton sequences. This direction is then used to determine if the generated point is in the hemisphere, and is optionally reversed. After this, the direction is adjusted using the user defined parameters: the x and y directions based on the maximum sampling area; the z by the tolerated depth difference between occluders.

Since normals obtained from the texture are in view space, the x and y coordinates of the sampling directions are actually the screen-space movements of the sampling point from the original pixel. Hence, no transformation is needed. The same goes for the depth values. I used the linear depth values stored in the alpha channel of the normal texture. The only transformation I used was adjusting the sampling area radius based on the depth value. This is necessary to have the computed indirect shadows invariant from the object's distance from the camera.

Screen-space ambient occlusion only computes a single value, so the application can store the darkening factor in a depth buffer. In a similar manner than in the case of screen-space directional occlusion the darkening factor is smoothed using bilateral filtering and optionally upsampled. The upsampled and smoothed darkening factor is then applied to the intensity texture computed in the direct illumination step.

This method requires a fairly large number of parameters: the sampling radius, the depth tolerance, the strength of the effect. These values are passed to the shaders using uniform variables. It is up to the user to set these parameters right for a given scene to get the best result.

5.5 High Dynamic Range Rendering

High dynamic range rendering includes rendering the scene in a high dynamic range (usually higher than the dynamic range of currently common displays), the consequent contrast reduction to be able to display them, and other miscellaneous methods trying to simulate real world camera effects, like the bloom effect. I decided to use Reinhard's operator and the bloom effect in the application as described in Section 2.2.

Generating high dynamic range images on the GPU is possible thanks to the support of floating point textures. Most graphics cards support nowadays 16bit and 32bit floating point textures, newer cards even 64bit ones. These textures allow storing values greater than one and also negative values, which is essential for computing the log-average gray. High dynamic range images are rendered in the application in the following way:

1. Render the intensity in the direct illumination step into a floating point texture.
2. Based on a threshold filter out high intensity pixels and store the difference from the threshold.
3. In the same shader convert the intensity for each pixel to grayscale. The output is an RGBA float texture, where the RGB data are the high intensity pixel colors and the alpha channel is the logarithm of the grayscale intensity of the pixel.
4. In order to get the bloom effect, apply a Gaussian blur on the texture from the previous step.
5. Compute the average gray value, by first generating mipmaps for the texture, and then rendering the texture using a full screen plane onto a small framebuffer (3x3 pixels). The pixels in the rendered framebuffer represent the average values of the pixels from the high resolution texture. Based on these values, compute the log-average gray.
6. Add the blurred high intensities to the intensities from the direct illumination step.
7. Scale intensities based on the image "key value" and the log-average gray value.
8. Apply the tone mapping operator based on equation (2.4).

In the fifth step I used only the pixel in the center to get the log-average value. Usually the region of interest is in the center of the image and scaling the intensities using this value might give better results for this region. As the high intensity pixels are blurred anyways, and the grayscale intensities are only used to get average, lower resolution can be used for their floating point texture. This improves speed and blurring is more effective.

The results of the tone mapping operator greatly depend on the image "key value", which should be a value between $(0.045, 0.9)$ indicating the subjective darkness or lightness of the image [18]. I also experimented with automatically adjusting the "key value" based on the average gray.

Chapter 6

Results

As mentioned earlier I implemented the application using SDL and OpenGL in the programming language C++. Additionally I relied upon the library GLEW[1] to handle OpenGL extensions. I also used the glm[2] library, which made matrix and vector operations easier on the CPU side of the application. The created application loads scene data from OBJ files using a freely available library[3]. However, I made a few bug fixes and modifications to the library, so the source code does not match exactly the version available publicly from the internet.

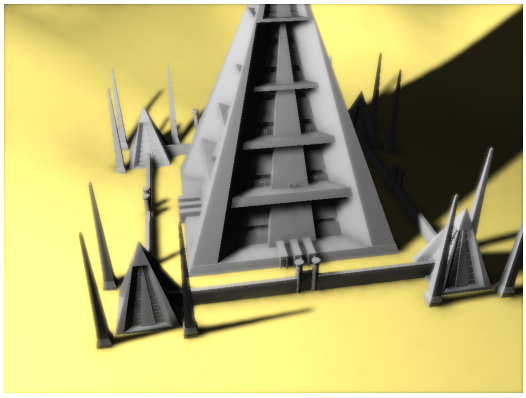
The final application consists from a single binary and the required data (models, shaders, textures and font). I tested the application on four models: a cornell box, a dragon[4] in a cornell box, pyramids and a temple (Figure 6.1). These give a good coverage over different types of scenes: small, large, with high number of faces and scenes with simple geometry. This enables to monitor how each method is effected by face count and scene complexity.

The application enables switching between methods during runtime. This is done by reinitializing all steps. The settings for methods between switches are naturally saved. The user can also control the camera and lights along with all parameters for the global illumination methods. The application informs the user about the changed values and the current frames per second using text overlay.

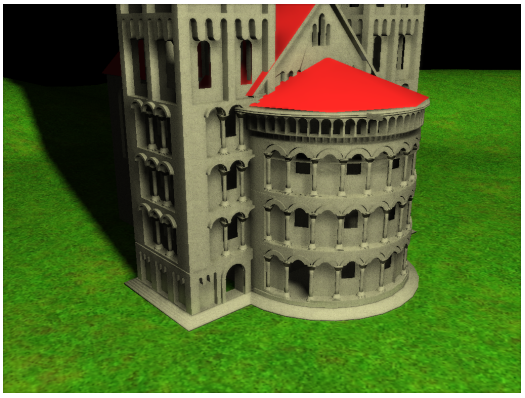
During initialization the application does not generate any new shaders for materials. All shader sources are static in the sense that they are not modified in any way by the application. Parameters for each method, lighting and material information is all passed through uniforms to the shaders. In order to avoid some code duplication, the implementation of some functions for different methods is pushed into separate shader source files. These shader parts are then linked together based on the currently enabled method. (For example shadow mapping techniques use the same interface, and the correct method's fragment shader is just linked together with the direct illumination fragment shader.)

The application was tested on an AMD ATI Radeon™ HD 4850 graphics card. If not specified otherwise, during the testing I used 1024x768 resolution. The resulting frames per second might not reflect the real speed of the used techniques. They hold information only relative to each other. In real world use the speed might also be affected by different things: the type of graphics card, load on different chips of the graphics card, used resolution,

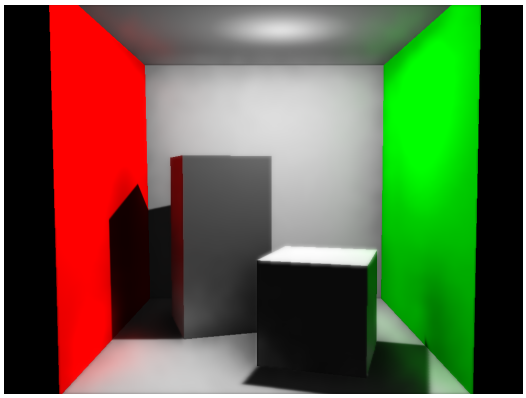
[1] http://glew.sourceforge.net/
[2] http://glm.g-truc.net/
[3] http://www.kixor.net/dev/objloader/
[4] The dragon modell comes from the Stanford 3D Scanning Repository.



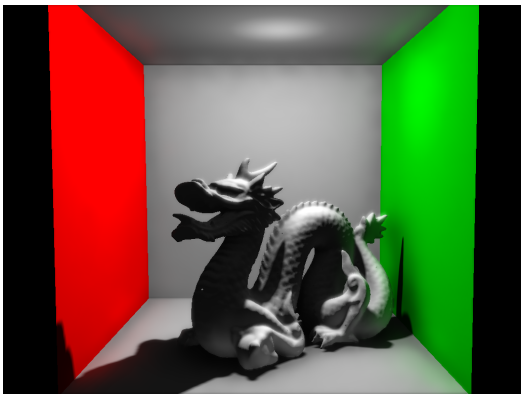
(a) Pyramids (∼16k faces)



(b) Temple (∼104k faces)



(c) Cornell box (30 faces)



(d) Dragon (∼200k faces)

Figure 6.1: The four scenes for testing. (All of them are rendered using shadow mapping with percentage closer filtering, screen-space directional occlusion and high dynamic range rendering.)

etc. The requirements against the quality of the resulting images have a great effect on render times, too. For example the degree of required smoothing and the precision of global illumination methods.

For running the application on different machines, at least OpenGL 3.2 support is necessary. Such systems are still not too common, so replicating the tests might not be easy. Therefore I focused more on the difference in quality of the images rendered with different methods and options, and used the frame rates only as an indicator for the cost in rendering time. In order to remove fluctuations in frame rates, I ran the tests several times and only the average frame rates are shown in graphs in the following sections.

6.1 Shadow Mapping

Shadow mapping is one of the basic methods to improve the rendered image of the scene to be more realistic (Figure 6.2). It helps the viewer to perceive more the 3 dimensions of the scene, so it does not look like a flat texture. I have implemented three types of shadow mapping methods in the application: simple shadow mapping, shadow mapping with percentage closer filtering (PCF) and variance shadow mapping (VSM). A comparison

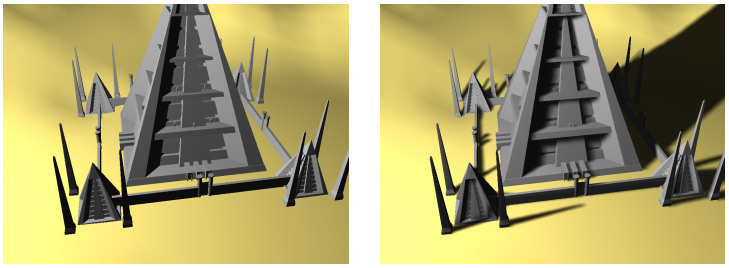
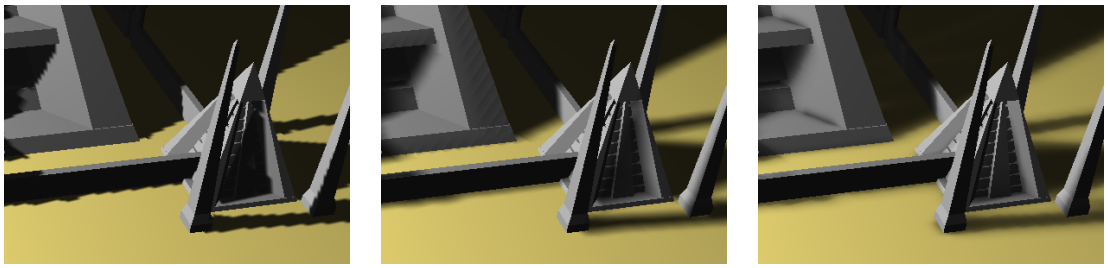


Figure 6.2: Pyramid scene without and with shadow mapping.

of the three methods is in Figure 6.3.



(a) Simple shadow mapping (b) PCF (c) VSM

Figure 6.3: Comparison of shadow mapping techniques. Left to right: simple shadow mapping, percentage closer filtering and variance shadow mapping.

Simple shadow mapping uses only one texture lookup. This results into jagged edges for shadows, because the depth texture has a limited resolution. The percentage closer filtered shadow is a bit better. The main disadvantage of these methods is the bias. Even though I used 32bit depth textures, for objects further from the light even a small bias can be disturbing. For simple shadow mapping I set the bias manually, so that no artifacts would appear. When using the same bias for percentage closer filtered shadow mapping, there are already dark lines on the large pyramid on the top of the image; and the shadows are already visible shifted to the right direction. In the case of variance shadow mapping, I actually did not use any bias and there are no lining artifacts. The only visible problem concerning bias is that the corners are darkened a bit too, similar to screen-space ambient occlusion, but only on lit surfaces. The main disadvantage of variance shadow mapping is light bleeding where the variance of depth values is high in the shadow map. It can be observed at the right bottom edge of the image. The shadow behind the second obelisk is much lighter. A better example will be in Figure 6.5. There are methods to fix this behavior, but since variance shadow mapping was not among the main goals of the project, I did not implement them. [1]

In Figure 6.4 is a chart showing the achieved frames per second for each method. I used only two scenes: the pyramid and the dragon scene. The results show, that the simple shadow mapping technique was the fastest. The extra 8 texture lookups and comparisons for percentage closer filtering cost about 20% (I used a 3x3 pixel kernel). Even though variance shadow mapping without blurring does only one texture lookup, it still came out slower than percentage closer filtering. I suspect the main reason for this is that I used 32

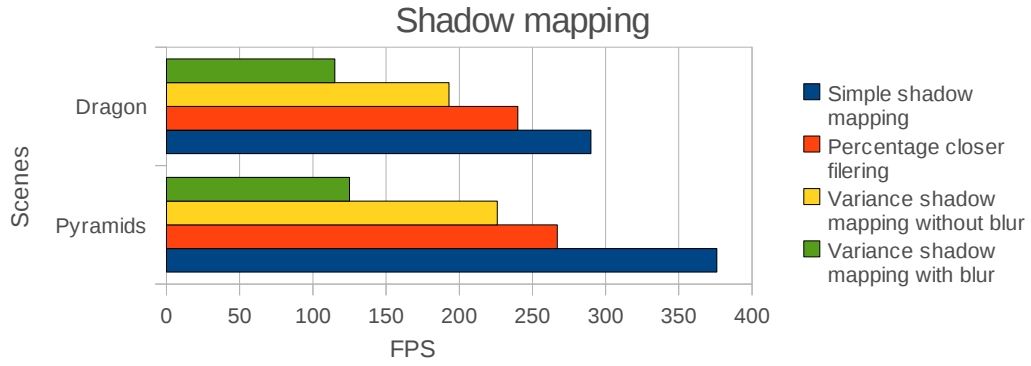
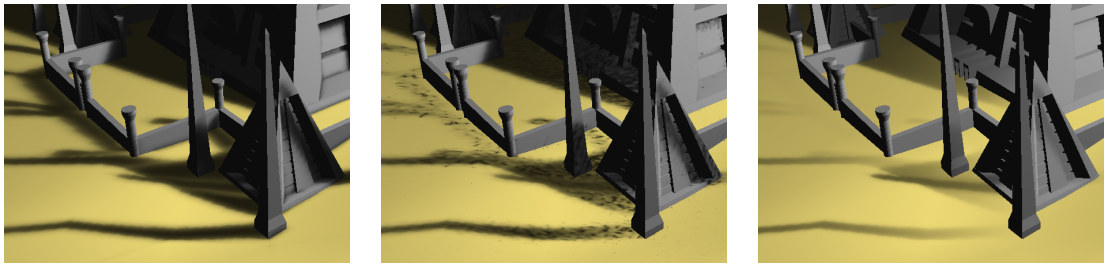


Figure 6.4: Speed of shadow mapping techniques.

bit floating point RGBA texture to store the depth and square of the depth. 16bit texture did not provide enough precision, but it was much faster (255 FPS without blurring and 167 with blurring for the pyramid scene). The huge difference in speed is probably due to the weaker graphics card. On newer hardware the difference would be much smaller. The lack of precision when using 16bit floating point textures caused artifacts due to the small variance, and also shadows close to objects were much lighter. The artifacts can be fixed using a higher threshold for variance (Figure 6.5).



(a) 32bit (b) 16bit (c) 16bit, fixed threshold

Figure 6.5: Variance shadow mapping depending on the type of the texture.

6.2 Screen-space Directional Occlusion

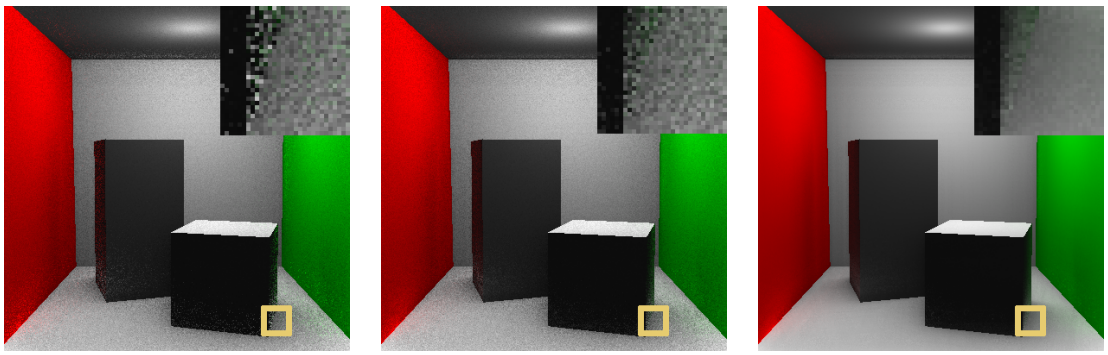
The images for the scenes in Figure 6.1 have already been rendered using the screen-space directional occlusion method with the proposed improvements. The effect of the method should be visible in the corners. Underneath the dragon the floor is much darker. On the wall close to the dragon's tail, between the dragon's legs, between the boxes on the cornell box there are indirect shadows, too.

The other main feature of screen-space directional occlusion, indirect bounce of light, might be observed in the cornell box on the side planes of the small boxes in the middle close to the red and green wall. Indirect bounces from the floor are also visible on the belly of the dragon; some green bounces are on the tail from the green wall. Green bounces from the grass can also be seen on the bottom of the left tower of the temple.

These all make the images much more realistic. In order to get a clearer picture how the proposed improvements and parameters effect the method, I tested screen-space directional

occlusion without any other method (high dynamic range rendering, shadow mapping). I would like to start from the original two-step method as a reference.

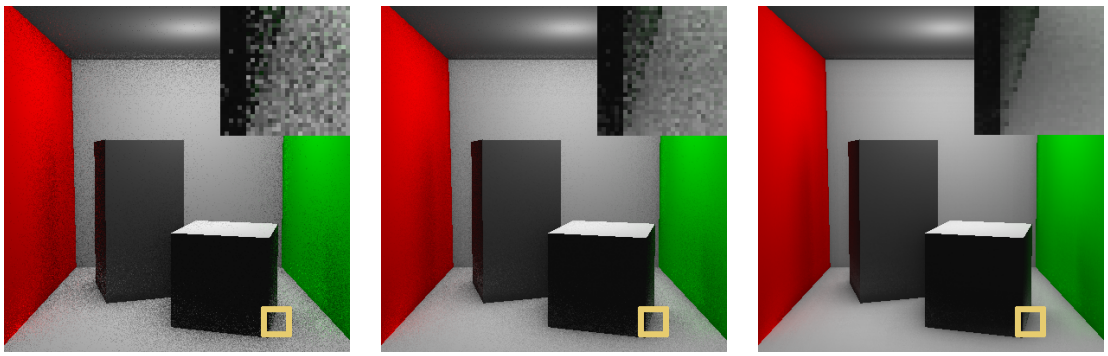
In Figure 6.6 are images with different sample count using the original two-step algorithm. The images were rendered with 400x400 resolution. Even with vast number of sampling points there is still some noise visible, and the frames per second dropped significantly. This concludes, that just increasing the number of the samples does not help after a certain point and smoothing might be more effective.



(a) 10 - 160 FPS (b) 30 - 115 FPS (c) 300 - 15 FPS

Figure 6.6: The screen-space directional occlusion method implemented using the original algorithm with different number of samples per pixel.

These images were already rendered using the Halton sequences for sampling. I did not notice any disturbing artifacts, and with growing number of samples the quality of images got better. For 10 sampling points there are some high intensity bounces. For 30 sampling points these already disappear. There is still some noise, but most of it is gone for 300 samples per pixel. In Figure 6.7 are images rendered using the proposed modulation factor and with the two steps of screen-space directional occlusion merged together.

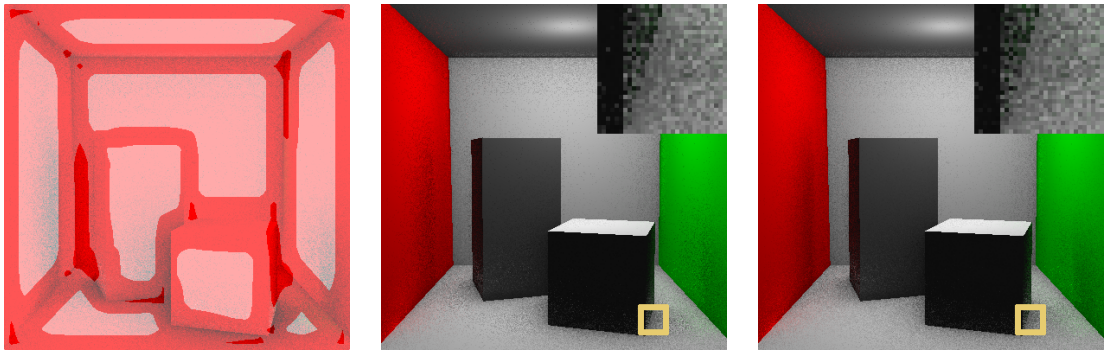


(a) 10 - 320 FPS (b) 30 - 160 FPS (c) 300 - 20 FPS

Figure 6.7: The screen-space directional occlusion method implemented using the modulation factor with different number of samples per pixel.

Again, images rendered with small number of samples are fairly noisy. The overall quality is comparable with the images rendered with the original method. The disadvantage of computing the bounce and darkening factor in one step is visible on the images with 300 samples. The darkening factor is used to darken the bounce too in order to avoid too bright

bounces in corners (on right top zoom-in the edge of the smaller box is darker). This makes the method less correct, but the speed improvement is massive especially for smaller number of samples, when using smoothing this problem is even less visible. The speed difference is mainly caused by the fact that for each sample the normal and depth are only read once from texture for each pixel in the modified version. In the two-step method they are read twice: once for computing direct illumination and the second time for the bounces.



(a) Number of sampling points (b) 10-30 - 200 FPS (c) 20 - 215 FPS

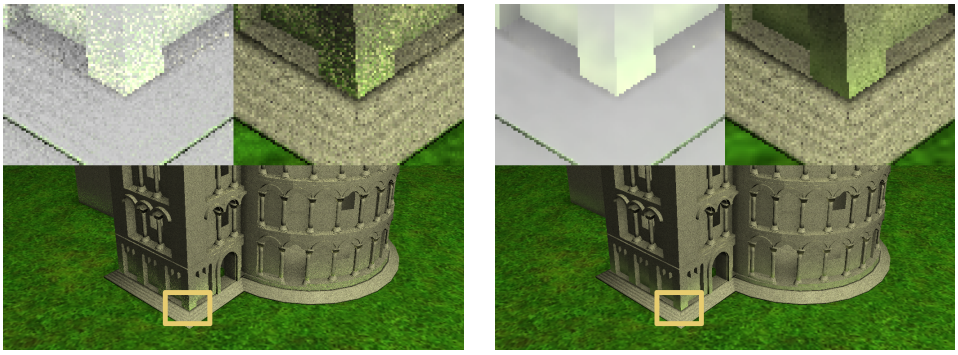
Figure 6.8: The left image shows the number of sampling points per pixel (10, 20 and 30). The darker red is the red channel for a pixel, the more sampling points were generated. In the center is the rendered image. For comparison there on the right is the image rendered using constant 20 samples.

In Figure 6.8 is demonstrated the algorithm when using preprocessing. The output with variable number of samples is comparable to the image rendered using constant number of samples. The method basically redivides the samples between different parts of the image. Even though there are quite large planes where fewer samples were generated, and only a few smaller regions where more samples were generated than the average, the render times did not improve. I can only guess that the reason behind it is the SIMD architecture of the GPU. With variable number of samples some processing units might still be computing, leaving the rest with less sampling points waiting. I do not think the preprocessing step has much of an impact on the speed, since it is computed in extremely low resolutions. (For 1024x768 original image with resolution 64x48 - 16x subsampling). The single texture read in the global illumination step does not cost this much speed next to the texture lookups for each sample to get normals, depths, intensities (approximately $20 \cdot 3 = 60$ texture lookups).

Figure 6.9 shows the results of the smoothing. Notice that only the indirect bounces and the darkening factor is smoothed, and it does not affect the texture on the walls. Applying the same effect for the results of the original screen-space directional occlusion, the intensities for each pixel, would blur the finer details, too. Since the smoothing is geometry aware, the bounces on the walls do not propagate to the stairs of the temple.

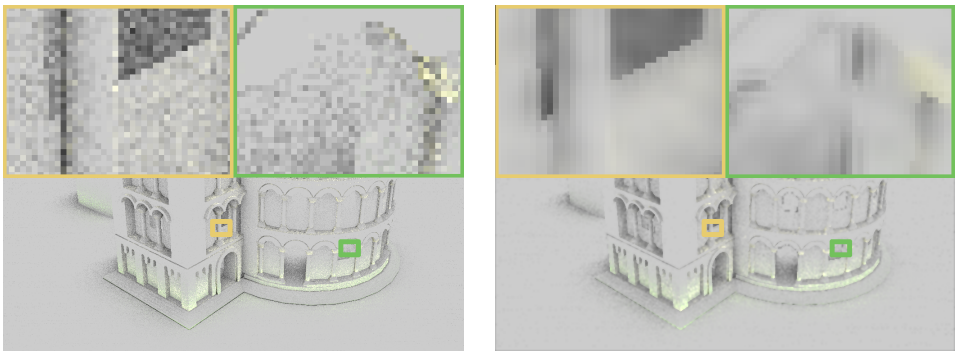
Unfortunately the smoothing is quite expensive. It costs around 20 % of the performance. The bounce and the darkening factor was stored in separate textures, so the application smoothed them separately; which is not the fastest solution either. Even though smoothing is not the most effective, it does remove the noise; especially high intensity bounces, which were the most disturbing.

Subsampling is demonstrated in Figure 6.10. When rendering the subsampled image no smoothing was used whatsoever, and thanks to subsampling the application ran twice



(a) Without smoothing - 40 FPS (b) Smoothed - 34 FPS

Figure 6.9: The temple scene with and without smoothing. For smoothing a 11x11 kernel was used and 20 sampling points for screen-space directional occlusion. The left top part of the images show the zoomed in region's darkening factor and bounce.

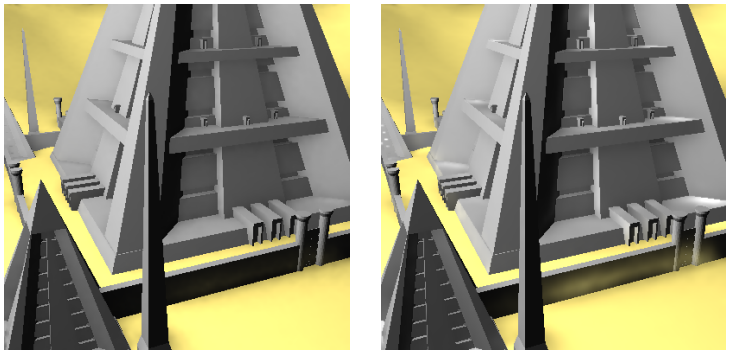


(a) Without subsampling and smoothing - 40 FPS (b) With subsampling - 80 FPS

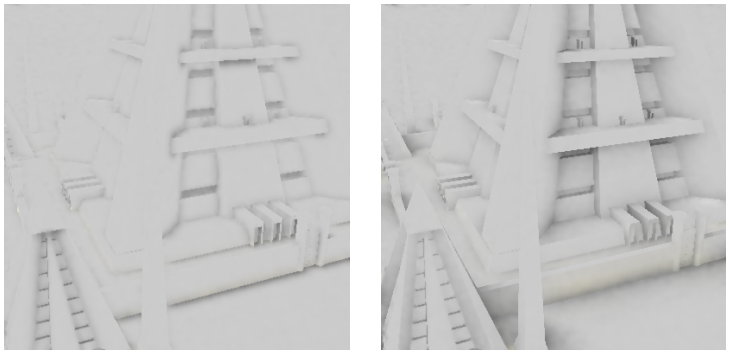
Figure 6.10: The result of subsampling demonstrated on the darkening factor. The right image was rendered using the quarter of the full resolution - 1024x768 for the full resolution and 512*384 for the screen-space directional occlusion and bounce.

as fast as without it. Since upsampling also adds a little blur, the smoothing kernels for bounce and the darkening factor might be smaller; lowering the render time even more. The joint-bilateral upsampling with a small 3x3 kernel is still sufficient to keep edges in the image sharp. The darkened border of a window showed in the top right corner is not caused by upsampling. It is caused by the normals. When computing screen-space directional occlusion in the subsampled resolution, the normals between pixels are interpolated changing the normal's direction and causing occlusion. These artifacts are fairly rare, and the gained speed-up is worth it.

Controlling the parameters for screen-space directional occlusion is quite straightforward. Figure 6.11 shows images with different bounce strength and the darkening factor for smaller and larger sampling radius. Setting these values right might depend on the scene. For small radius the smaller geometry details are nicely visible. For larger radius these fine details are lost, but the rougher object shape gets highlighted more. In the future experimentations can be done to combine the results rendered with different sized sampling radiuses in order



(a) weak bounce (b) strong bounce



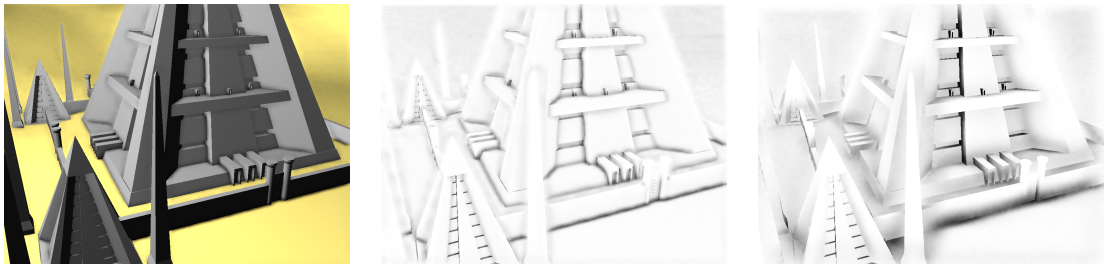
(c) small radius (d) large radius

Figure 6.11: Different settings for bounce strength and maximum sampling radius. (Images were rendered with 400x400 pixel resolution.)

to get a better approximation for global illumination. An alternative solution might be the following: during sampling, first sample with smaller radius. With growing number of samples, increase the size of the sampling radius. The darkening caused by more distant objects should be set smaller, so that they do not interfere too much with the fine details. At least in theory the same weight would make larger constant indirect shadows (they will not get darker deeper into the corner).

6.3 Screen-space Ambient Occlusion

The main reason behind including screen-space ambient occlusion was to provide a solution to which the screen-space directional occlusion could be compared to. The implementation is not the most optimized. I used the same sampling method as with screen-space directional occlusion so as to avoid speed difference caused by different methods. In order to get better results, I tried to find a good balance between the darkening caused by different normals and the penalty for depth difference. The results can be seen on Figure 6.12. From the images it is clear that for small radius the method works reasonably well, but with larger radius some problems appear. For example behind the obelisk in the front, the darkening on the wall is inconsistent with other parts of it. Same goes for the bottom of the large pyramid behind the obelisk. Another artifact I could not remove completely is the silhouette around objects.



(a) (b) small radius (c) large radius

Figure 6.12: Screen-space ambient occlusion. The two left images show the darkening factor for different sampling radiuses.

I also added some of the modifications I used for screen-space directional occlusion: smoothing and subsampling. The penalty for smoothing and the speedup for subsampling were similar. Without subsampling and smoothing the method runs on the pyramids scene with 125 FPS. With added smoothing (11x11 kernel) it decreased to 100 FPS and with subsampling and smoothing on, using quarter of the full resolution, it went up to 175 FPS.

6.4 Comparison of Global Illumination Methods

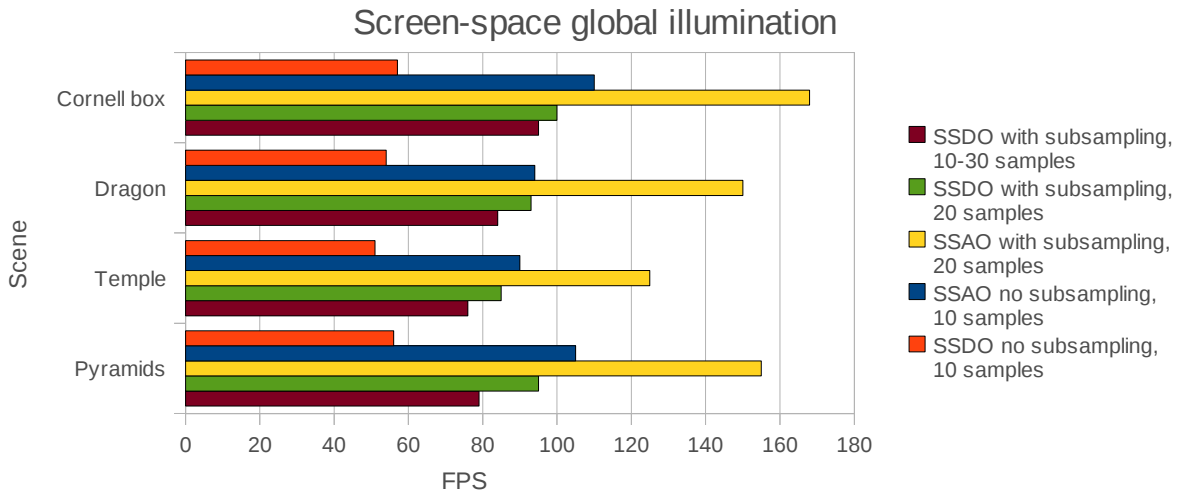
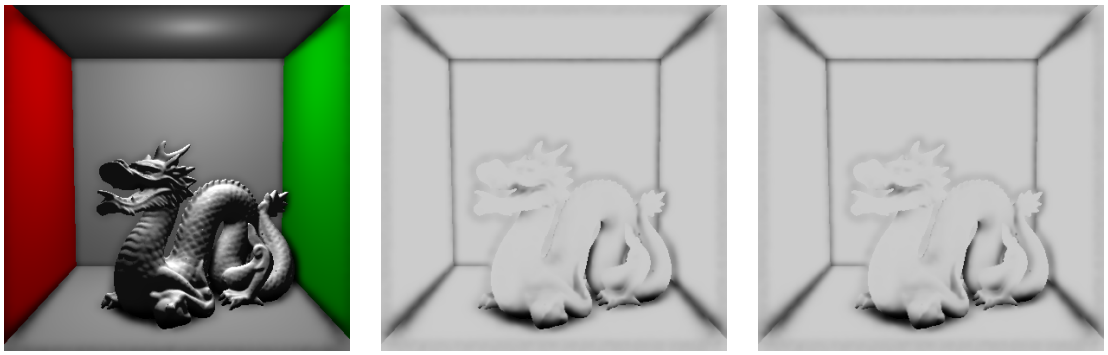


Figure 6.13: Graph showing the frames per second for the implemented methods: SSAO - screen-space ambient occlusion, SSDO - screen-space directional occlusion. (upsampling kernel size 3x3; 11x11 smoothing; 1024x768 full resolution; 512x384 for the subsampled image)

The results of the achieved frame rates are summarized in Figure 6.13. The methods were tested on all four scenes with resolution 1024x768. However, these frame rates are just exemplary. The speed of these methods depends also to a great extent on the resolution, graphics hardware (especially floating point texture performance), as well as the the degree of required smoothing.

Screen-space ambient occlusion was naturally the fastest. The additional computations and texture lookups to get bounces for screen-space directional occlusion makes it almost twice as slow. Probably the manipulation with floating point textures for intensity has an effect on this, too (screen-space ambient occlusion uses only the depth and normal textures). From the measured frame rates it is clear that the limiting factors are the speed of fragment shaders and the number of texture lookups per pixel. These are the areas that should be more optimized in the future.

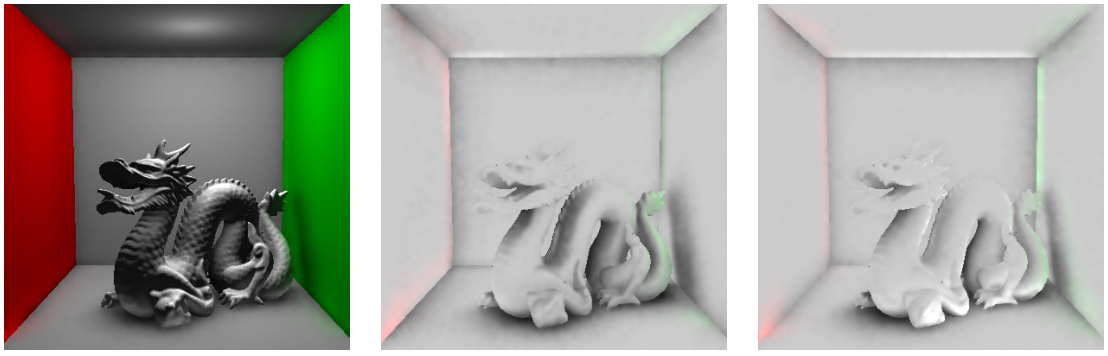
Due to the fact that both methods are computed in the screen-space, a few problems arise. The lack of complete knowledge of 3D causes occluders not visible from the camera's point of view to be discarded, hence making the results highly view-dependent. For more complex scenes even a small change in camera position may reveal parts of the scene previously hidden and cause new shadows and indirect bounces. The authors of the screen-space directional occlusion method suggested using depth peeling for partly solving this problem. Storing multiple depth values for each point in multiple passes gives more information on the scene structure; however, it makes the speed of the technique dependent on scene complexity. It also further slows down occlusion computations by forcing it to read values from multiple buffers and calculating the depth test for each. Alternatively, the authors also suggest using multiple viewpoints. This in theory could give better results than depth peeling, but correct positioning of the cameras may vary based on the scene type.



(a) SSAO

(b) light in the top left back corner

(c) light in the top right front corner



(d) SSDO

(e) light in the top left back corner

(f) light in the top right front corner

Figure 6.14: Screen-space ambient occlusion versus screen-space directional occlusion with different light positions. (The darkening factors are multiplied with 0.8, so that the bounces are visible, too.)

A comparison of the two methods can be seen in Figure 6.14. The difference between the two techniques can be easily observed on the darkening factor around the tail of the dragon. While screen-space ambient occlusion darkens only the silhouette, the tail in screen-space directional occlusion darkens the wall close to it in 3D. The next visible difference is when the light position is changed. While screen-space ambient occlusion remains static with moving light, the shadows and bounces caused by occlusion in screen-space directional occlusion move a little in the opposite direction. The difference is most visible on the back and the belly of the dragon. Thus the additional bounce for screen-space directional occlusion makes the images much more believable.

(a) SSDO (b) Bounce + darkening factor (c) Darkening factor

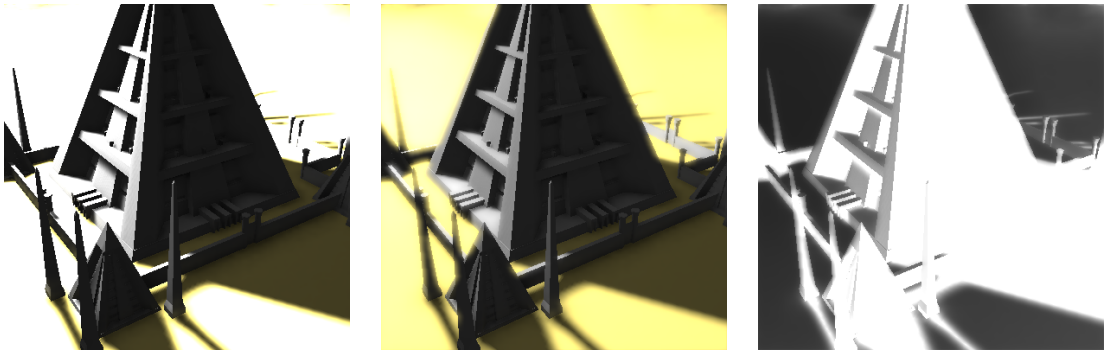
Figure 6.15: The result of screen-space directional occlusion for three light sources. A red light is at the front middle right edge, a green light is in the front top left corner and a blue light is in the back top left corner of the cornell box.

Unlike screen-space ambient occlusion, screen-space directional occlusion can handle also multiple light sources with different colors. Screen-space ambient occlusion is just static the same way as on Figure 6.14. Screen-space directional occlusion can on the other hand work with one darkening factor per color channel. The Figure 6.15 shows the result of the method with three different light sources. Notice that the darkening factor leaves almost all red on the back of the dragon, since there is no occluder between the back and the red light source; but it does remove all green, because the neck of the dragon does not allow any light to pass from the green lights direction. The opposite effect can be observed under the neck of the dragon, where the green color is kept and the red is removed.

6.5 High Dynamic Range Rendering

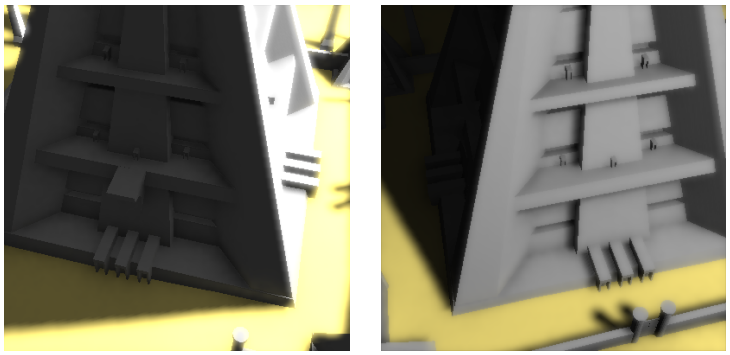
High dynamic range rendering can really improve the image quality to simulate real-world cameras. Figure 6.16 shows images rendered with the application. The light was set very bright, so that without high dynamic range rendering lit surfaces burn out into white. The only recognizable details are mostly in shadows. With high dynamic range rendering the colors and details reappear for regions with high intensity. The bloom effect is visible for example on the obelisk on the left. Due to the bright sand in the background, the edges of the obelisk almost disappear.

With the method I implemented for high dynamic range rendering I was able to manually set the threshold intensity over which colors are clamped to white. The application also tries to set the key value for tone mapping automatically. Without automatically scaling the



(a) No HDR - 175 FPS (b) HDR - 150 FPS (c) Scale

Figure 6.16: Results of high dynamic range rendering (HDR). Left without HDR in the middle with HDR. The right image shows intensity scale for each pixel (400x400 resolution, screen-space directional occlusion, shadow mapping with percentage closer filtering)



(a) 0.22 (b) 0.6

Figure 6.17: Automatic key value set based on the log-average gray intensity. Under the images is the used key value for the tone mapping.

key value bright parts of the scene would be too dark, and dark parts too light. Figure 6.17 shows the adaptiveness of this tone mapping operator. On the left image the tone mapping operator makes the lit side of the pyramid look quite bright. For this image the average gray intensity is smaller, so the key value is set lower. When moving the camera in front of the pyramid, the average intensity is much higher, so the key value is set higher, too.

The speed penalty for high dynamic range rendering with combination of screen-space directional occlusion is around $10 - 15\%$, but it depends on how stressed is the fragment shader by the global illumination method, and how precise floating point textures were used. Most of the performance is lost due to the Gauss blur for the bloom effect. The texture for bloom for this reason in the application was already subsampled. The results show that the small penalty is well worth it. High dynamic range rendering with bloom makes rendered images much more realistic.

Chapter 7

Conclusion

In this project methods for realistic lighting were presented. Shadow mapping is one of the most well known methods to achieve direct shadows from point and directional lights. Apart from the basic algorithm two more methods based on shadow mapping were implemented: percentage closer filtered soft shadows and variance shadow mapping. Both of them have advantages and disadvantages, but variance shadow mapping was much slower; mostly due to the usage of floating point textures.

In order to simulate real-world camera properties and to be able to display images with large contrast ratios, high dynamic range rendering was used. Compared to the global illumination methods, the cost of the simple tone-mapping and bloom effect was quite small.

For global illumination two methods were described. Screen-space ambient occlusion is a very fast approximation of ambient occlusion, but it has some limitations. Screen-space directional occlusion includes two generalizations that add directional occlusion and diffuse indirect bounces. Both extensions improved realism considerably.

In this project, a few experiments were made to the screen-space directional occlusion. Sample point generation based on the Halton sequence is an easy way to get uniform distribution of the sampling points. The periodic properties of the Halton sequence can also be used to potentially further optimize the method. This was exploited to generate a variable amount of sampling points, but still have a pseudo-uniform distribution and a full coverage of the hemisphere. In order to reduce noise, a modified version of bilateral filtering was used, which took into account the geometry information as well to avoid color bleeding over edges and between objects. Both global illumination methods were computed in lower resolutions to speed up the methods. For upsampling to the original resolution, joint bilateral upsampling was used to honor geometry properties.

Screen-space directional occlusion was notably slower than screen-space ambient occlusion; but the additional features, like more precise approximation of ambient occlusion, indirect bounce, dependency on light position and color, make the resulting images way more realistic. In the future, more experiments can be made to both global illumination methods. For example the usage of more viewpoints or depth peeling to get better approximation of the scene; the combination of darkening factors with different sampling radiuses; merging lighting from point lights with lighting from environment maps and other extension and improvements could be explored.

Bibliography

[1] L. Bavoil. Advanced soft shadow mapping techniques. [online]. `http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_SoftShadowMapping.pdf`.

[2] L. Bavoil, M. Sainz, and R. Dimitrov. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, page 22:1, New York, NY, USA, 2008. ACM.

[3] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *SIGGRAPH Comput. Graph.*, 15:307–316, August 1981.

[4] R. Dimitrov. Cascaded shadow maps (whitepaper). NVIDIA OpenGL SDK 10 Code Samples, 2007.

[5] K. Dmitriev, S. Brabec, K. Myszkowski, and H.-P. Seidel. Interactive global illumination using selective photon tracing. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 25–36, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[6] W. Donnelly and A. Lauritzen. Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 161–165, New York, NY, USA, 2006. ACM.

[7] F. Durand and J. Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 257–266, New York, NY, USA, 2002. ACM.

[8] P. Dutré. *Global Illumination Compendium. Technical report, Computer Graphics.* Department of Computer Science, Katholieke Universiteit Leuven, 2003.

[9] P. Dutré, K. Bala, and P. Bekaert. *Advanced global illumination.* Ak Peters Series. AK Peters, 2006.

[10] D. Filion and R. McNaughton. Effects & techniques. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 133–164, New York, NY, USA, 2008. ACM.

[11] F. Houlmann and S. Metz. High dynamic range rendering in opengl. [online]. `http://transportergame.googlecode.com/files/HDRRenderingInOpenGL.pdf`.

[12] A. Kaplanyan and C. Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH symposium on*

Interactive 3D Graphics and Games, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM.

[13] J. Kopf, M. F. Cohen, D. Lischinski, and M. Uyttendaele. Joint bilateral upsampling. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[14] M. Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA, 2007. ACM.

[15] G. Nichols, J. Shopf, and C. Wyman. Hierarchical image-space radiosity for interactive global illumination. page 1141–1149, 2009.

[16] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 283–291, New York, NY, USA, 1987. ACM.

[17] C. Reinbothe, T. Boubekeur, and M. Alexa. Hybrid ambient occlusion. EUROGRAPHICS 2009 Areas Papers, 2009.

[18] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. Photographic tone reproduction for digital images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 267–276, New York, NY, USA, 2002. ACM.

[19] T. Ritschel, Th. Grosch, and H.-P. Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 75–82, New York, NY, USA, 2009. ACM.

[20] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 73–80, New York, NY, USA, 2007. ACM.

[21] M. Stamminger and G. Drettakis. Perspective shadow maps. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 557–562, New York, NY, USA, 2002. ACM.

[22] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proc. IEEE Int. Conf. on Computer Vision*, pages 839–846, 1998.

[23] J. Vanek. *Měření výkonnosti grafického akcelerátoru, diplomová práce.* FIT VUT v Brně, Brno, 2010.

[24] L. Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 270–274, New York, NY, USA, 1978. ACM.

[25] S. Zhukov, A. Inoes, and G. Kronin. An ambient light illumination model. In *proceedings of the Eurographics Workshop in Vienna, Austria*, Rendering Techniques '98, pages 45–56, Springer, 1998.

Appendix A

Contents of the DVD

- *sources* – Source code with program documentation for application with Visual Studio 2008 project files and linux Makefile.
- *bin* – Compiled binaries with the necessary dynamically linked libraries for Microsoft Windows system.
- *doc* – Documentation with full size images and poster.
- *videos* – Videos demonstrating the methods.
- `README`